UNIVERSITAT POLITÈCNICA DE CATALUNYA

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Escola Superior d'Enginyeries Industrial,
Aeroespacial i Audiovisual de Terrassa

ANALYSIS AND STUDY OF A

SHALLOW WATER MODEL CODE FOR APPLICATIONS TO PLANETARY ATMOSPHERES

A thesis submitted by Arnau Prat Gasull for the BSc Degree in Aerospace Vehicle Engineering

June 10, 2018

Directed by:
Manel Soria Guerrero

Co-director:
Enrique García Melendo

Special thanks to:
Arnau Sabatés Urgell

“I would like to thank Manel Soria and Enrique García for their invaluable support during the course of this project and guiding through the journey. I could not have been introduced to atmospheric sciences better. I would also like to write a message to Arnau Sabatés: ‘look at what we have done!’”

To my parents, sister and friends...

Contents

List of Figures

Part I

General introduction

CHAPTER 1

Before reading

The reader (you) should note that this is a technical document which requires minimum knowledge on mathematics, physics and programming in order to be fully understood. The thesis is written for the average engineering student with interests in the topics aforementioned in the Table of Contents.

The author (me) recommends reading the work of Arnau Sabatés[8], a fellow student and a member of the research group. His Final Year Project (FYP) introduces the shallow water model with emphasis on the physics and real life phenomena. By reading the present document, the author assumes that the reader is familiar with this subset of the Navier–Stokes equations (NSE) and thereby, he presents and uses concepts with no previous detailed explanations.

1.1 To those students who will be reading this in the future

A personal note from the author:

"I hope you are as interested in planetary atmospheres as I am at the time of writing this thesis. This is the culmination of my bachelor's degree but I really cannot point out if I am more proud for this achievement or the results of this project. See, during the second year of my bachelor's I decided that my FYP would be about cloud simulation and that I would fully devote myself to this task. I even talked about it with friends and family, but they did not find it as cool as me. After I saw that it was quite difficult to pursue this project in Terrassa I abandoned the idea and hoped that I would find an interesting project in the list of FYPs when the time was to come. It was at the end of the 3rd course when I met Manel Soria in an optional subject about Message Passing Interface (MPI) programming. He introduced me to the NSE solvers as he felt my interest in this type of software. While I was not sure at first I gave it a try and unsuccessfully programmed my small solver for the NSE. At the arrival of Enrique García (last year of my bachelor's), him and Manel Soria decided to offer a FYP on planetary atmospheres and I was quickly informed of the project. I immediately applied for the project and was absolutely delighted to be accepted: my original idea was about to become a reality. Manel Soria and I started attending atmospheric sciences classes imparted by Enrique García at the start of October. Soon after, Arnau Sabatés joined the group and it was clear that we would be a team for the remaining months of both his and my bachelor's degree.

The experience has been extremely rewarding, even if great difficulties were encountered. The results after 5 months of intensive work are very satisfactory and match the expectations. It is no surprise that I have been short of time and the potential of the model's implementation is not truly shown in this document: I did not want this project to end. I have found a very attractive field of study and I will try to dive deeper into the subject even when I am no longer affiliated with the university. If you are unsure if you are interested on this matter, I suggest you read as much as possible on the topic and meet with any of the members of the research group."

1.2 Technical description of the document

This document contains an important number of links to the source code of Shallow Worlds when the appendices are part of the same file. If this is a version without the appendices, a lot of the desired functionality has been lost and the reader will have to manually search the function in the source code.

The source code of Shallow Worlds contains comments that can be processed by TEX. When the code and the text are included in the same file, every macro function mentioned in the contents becomes a link to the source code. The behaviour of the Shallow Worlds source code when included in the report is similar to that of `demo_sum(...)` (click it!) in this document and, as the reader will see when clicking the name of the function, is very convenient to a Shallow Worlds user.

```
1 double demo_sum(double a, double b) {
2
```

`demo_sum(...)` revised on 08/06/2018 by Arnau Prat Gasull.

Description
Returns the sum of $a + b$, where `a` and `b` are inputs of the functions.

Parameters
Below follows a brief description of the variables:
`double a` (input) The first number of the sum.
`double b` (input) The second number of the sum.

Notes
This is a demonstration test.

```
3 return a+b;
4 }
```

By including the source code in this memory, this document automatically becomes a reference guide for the current version of Shallow Worlds and sets the comment style guides for future development.

Moreover, this document contains animations that can be played under Adobe Acrobat. In other PDF viewers, the reader will see a still image. All the animations in this document can be made available under request.

CHAPTER 2

Overview

This report is part of the author's final thesis for the bachelor's degree in Aerospace Vehicle Engineering, undertaken at ESEIAAT—UPC and the outcome of a five month research task. The work group, constituted by Enrique García Melendo, Manel Soria Guerrero, Arnau Sabatés Urgell, and the author, has studied the shallow water equations for applications to planetary atmospheres and successfully developed a solver for the equations.

In this document, the reader will find the implementation notes of Shallow Worlds, the software referenced above. The physics behind the shallow water model have not been explained here, but in the work of Arnau Sabatés[8]. The interested reader will find precise information of the physics and a implementation of the model written in Matlab, which has been used to verify Shallow Worlds.

These theses are understood as manuals for futures students who want to dive into climate modelling, atmospheric flow simulations and numerical methods. The members of the research group are open to student applications for continuing the work presented in these FYP.

2.1 Aim

The aim of this project is to program and build a solver for the shallow water equations, a set of equations that are used in simulations of atmospheric phenomena. The author's intent is to efficiently produce results that closely match the observations and documented cases of storms and planetary flow.

In the long term view, this work will set the bases for an improved solver, which will account for convection and other variations of the variables in the z axis, as well as global dynamics and new phenomena.

2.2 Scope

This project is devoted to the simulation of planetary atmospheres and to the numerical integration of the shallow water equations. A one-layer shallow water model will be implemented, but the program will also be set up to support multilayer features with little modifications. The study of the physics behind the model is closely related to the use of the solver and therefore, the project must follow the trends in the planetary science in terms of file formats, algorithms...

Hydraulic engineering and other disciplines are definitely out-of-scope, even though the core of the solver may be suitable for applications other than atmospheric simulations.

2.3 Requirements

This solver will be used by academics and professionals for the study of planetary phenomena, so these users must be familiar with the structure of the program with little interaction. In fact, the source code must be

clear and good programming practices have to be employed. In order to create a proper solver, this should be written in C and MPI to ensure that the results will be computed and delivered as fast as possible on whichever infrastructure it is executed on, including supercomputers. The program must be written on top of a set of macros and functions that use MPI features, so it is as scalable and as versatile as possible.

The software must work with as many structured grids as possible to ensure reusability. In this direction, the core of the developed solver must not assume any coordinate system. Despite the fact that the domain may take almost any form, the solver shall be delivered with, at least, simple post-processing features that will accurately represent the information. This results have to be clear and easy to understand.

2.4 Justification

The original Shallow Worlds, written by Enrique García Melendo, cannot simulate large portions of the atmosphere efficiently. Because larger domains want to be studied, more computational power is required and by using MPI, the new Shallow Worlds should outperform the original solver. From now on, the solver coded by Enrique García will be referenced as the legacy Shallow Worlds, to denote the research group's goal to have a faster solver.

Part II

Shallow Worlds: development, results, validation and performance

CHAPTER 1

A brief introduction to planetary atmospheres

Weather forecasting has always been of great interest to human societies. It comes as no surprise, as a number of vital activities such as agriculture are highly dependent on weather conditions. Weather prediction has come a long way since the first attempts and this can be observed in the precision of the forecasts. In this sense, accurate 7-day forecasts are nowadays available to the general public on demand. Today's forecasting methods make use of supercomputers and clusters, which process data received from weather satellites, weather stations and other monitoring systems.

With the rise of the computing power more complex and accurate models have been developed to the point that current models take into account a great number of variables. However, while Earth's atmosphere is being studied and simulated with detail, atmospheric phenomena of other celestial bodies cannot be modelled [in general] with the same degree of accuracy as that of the Earth's, be it for the importance to human societies, the available data or the knowledge.

Atmospheric physics studies the atmosphere with the use of the NSE, advanced chemical models, radiative transfer processes and statistics, for example. Forecasting of weather phenomena of other planets is heavily dependent on available data and thereby, on observations from Earth of distant objects and costly space missions. Despite the difficulties, progress has been done in the study of atmospheres of other planets and different models are already available. A model that has been used to simulate phenomena in Saturn's atmosphere[4] with reasonable precision is known as the shallow water model, and it is the object of this work.

The shallow water equations are a subset of the NSE and are suitable for simulations as the hypotheses that are made match the characteristics of planet atmospheres. From now on, these are assumed to be thin with respect to the radius of the planet and therefore, hydrostatic. Because the layer is so shallow, movements of masses only happen in "two dimensions". The shallow water equations used in the simulation of planetary atmospheres are derived below.

CHAPTER 2

A presentation of the shallow water model

The shallow water model is a subset of the NSE which is used in a variety of fields, from hydraulics engineering to, as the reader has seen, atmospheric sciences. A good understanding of the physics behind the equations is vital in order to carry out simulations and post-process the results.

2.1 Hypotheses

The flow is assumed to be incompressible, so, from the continuity equation, $\nabla \cdot \vec{u} = 0 \implies \nabla \cdot (k\vec{u}) = \vec{u} \cdot \nabla k + k\cancelto{0}{\nabla \cdot \vec{u}}$. The flow is also supposed to be inviscid, so the forces originated from viscous sources will be disregarded.

2.2 Fundamental equations

The shallow water equations can be derived from the continuity equation, the NSE and the thermodynamic equation. This set of equations is presented in (2.1), where ρ is the fluid density, p is the pressure, ϕ is the gravitational potential, $\psi = \psi(\vec{u})$ is the viscous force per unit volume, T is the temperature, C_p is the specific heat at constant pressure, κ is the thermal conductivity, and Q is the addition of heat per unit mass and time. Observe how $\frac{d}{dt} = \frac{\partial}{\partial t} + (\vec{u} \cdot \nabla)$ is the material derivative.

$$\begin{cases} \dfrac{\partial \rho}{\partial t} + \nabla \cdot (\rho \vec{u}) = 0 \\ \rho \dfrac{d\vec{u}}{dt} = -\nabla p + \rho \nabla \phi + \psi \\ C_P \dfrac{dT}{dt} - \dfrac{1}{\rho} \cdot \dfrac{dp}{dt} = \dfrac{k}{\rho} \nabla^2 T + Q \end{cases} \tag{2.1}$$

The set of equations (2.1) may be complemented with the ideal gas equation $p = \rho RT$, with R being the specific gas constant.

If the reference frame is a non-inertial rotating one, the conservation of the momentum equation—the second line in (2.1)—is rewritten as (2.2).

$$\frac{\partial \vec{u}}{\partial t} + (\vec{u} \cdot \nabla)\, \vec{u} + 2\vec{\Omega} \times \vec{u} = -\frac{\nabla p}{\rho} + \nabla \phi + \frac{\psi}{\rho} \tag{2.2}$$

In equation (2.2), ϕ is the effective gravitational potential, which accounts for the centripetal acceleration $\vec{\Omega} \times (\vec{\Omega} \times \vec{r})$. For our purposes, the effective gravitational potential is such that yields $\nabla \phi = -g\boldsymbol{k}$. From now on, g is the value of the gravitational potential that includes centrifugal acceleration.

The viscous forces are negligible outside the boundary layer $\psi \approx 0$.

2.2.1 Demonstration of the geostrophic equilibrium equation

Assuming the horizontal velocities u and v is U are of the same order of magnitude, their characteristic magnitudes can be represented by U. The characteristic magnitude of the length of the horizontal scale of movement is L and that of the duration of the phenomena is T. A variable of interest commonly used in atmosphere applications is the Rossby number $\mathrm{Ro} = \frac{U}{fL}$ which compares the inertial acceleration and the Coriolis acceleration. Note that it is supposed by convention that $T \sim \frac{L}{U}$.

$$\left|\frac{\partial\vec{u}/\partial t}{2\vec{\Omega}\times\vec{u}}\right| = \frac{U/T}{2\Omega U} = \frac{U^2/L}{2\Omega U} = \frac{U}{2\Omega L} = \mathrm{Ro} \tag{2.3}$$

In inviscid rotating fluids it may be seen that $|\partial\vec{u}/\partial t| \ll \left|2\vec{\Omega}\times\vec{u}\right|$. Therefore, $\mathrm{Ro} \ll 1$ and equation (2.2) becomes equation (2.4).

$$2\vec{\Omega}\times\vec{u} = -\frac{\nabla p}{\rho} + \nabla\phi \tag{2.4}$$

As seen in Figure 2.1, $\vec{\Omega}$ is aligned with the rotation axis of the planet, so for any point on the surface $\vec{\Omega} = \Omega\cos(\varphi)\boldsymbol{j} + \Omega\sin(\varphi)\boldsymbol{k}$, while $\vec{u} = u\boldsymbol{i} + v\boldsymbol{j} + w\boldsymbol{k}$. Therefore, equation (2.4) is written in component form as (2.5).

$$\begin{pmatrix} -2\Omega v\sin\varphi + 2\Omega w\cos\varphi \\ 2\Omega u\sin\varphi \\ -2\Omega u\cos\varphi \end{pmatrix} = \begin{pmatrix} -\frac{1}{\rho}\frac{\partial p}{\partial x} \\ -\frac{1}{\rho}\frac{\partial p}{\partial y} \\ -\frac{1}{\rho}\frac{\partial p}{\partial z} - g \end{pmatrix} \tag{2.5}$$

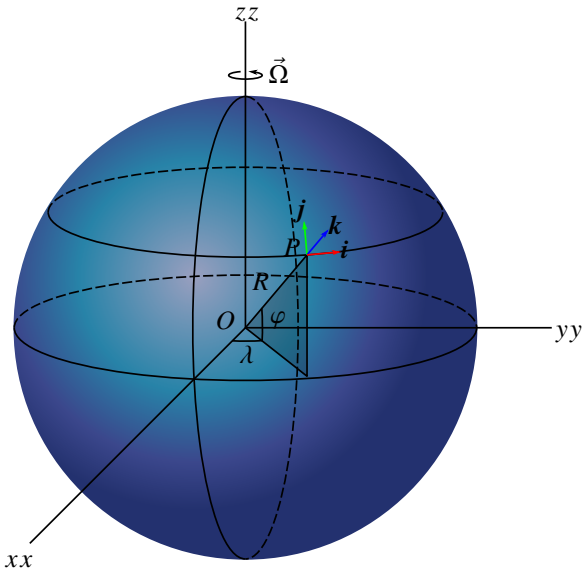


Figure 2.1: Reference systems

If the atmosphere is considered as a thin layer of fluid with $D/L \ll 1$, where D is the characteristic thickness and L is the characteristic horizontal length, it must be satisfied that $w \ll u, v$ to ensure geometric consistency. This can be seen in Figure 2.2: because $T = D/W = U/L$, equation (2.6) is demonstrated.

$$W = \frac{D}{L}U = \delta U \tag{2.6}$$

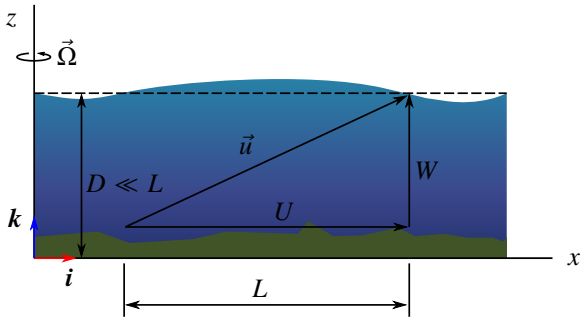


Figure 2.2: Representation of $\vec{u}$

Assuming that the atmosphere is also stratified and in hydrostatic equilibrium for analysis of large scale phenomena, (2.5) can be rewritten as (2.7). Because the horizontal movement is orders of magnitude greater than the movement in the vertical axis, the hydrostatic assumption is an accurate approximation.

$$\begin{pmatrix} -2\Omega v \sin\varphi \\ 2\Omega u \sin\varphi \\ 0 \end{pmatrix} = \begin{pmatrix} -\frac{1}{\rho}\frac{\partial p}{\partial x} \\ -\frac{1}{\rho}\frac{\partial p}{\partial y} \\ -\frac{1}{\rho}\frac{\partial p}{\partial z} - g \end{pmatrix} \tag{2.7}$$

Note that

$$f = 2\Omega \sin\varphi \tag{2.8}$$

is the Coriolis parameter or the planetary vorticity and from the first two equations in (2.7) it is easy to derive the expression (2.9), which is representative of the *geostrophic equilibrium*. This causes e.g. the counter-clockwise movement of air masses around a depression on the northern hemisphere of the Earth.

The geostrophic equilibrium

$$\begin{pmatrix} u \\ v \end{pmatrix} = \frac{1}{f\rho}\boldsymbol{k} \times \nabla p \tag{2.9}$$

2.2.2 The shallow water model

The shallow water model[7] may be derived from the situation represented in Figure 2.3.

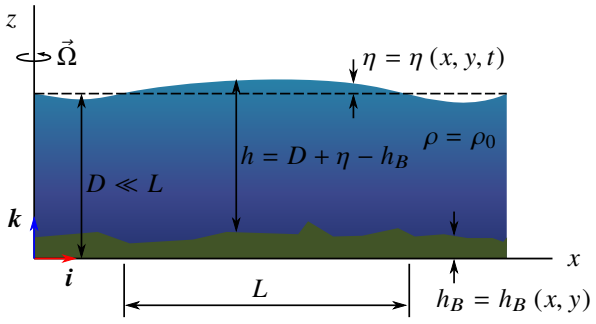


Figure 2.3: Variables of interest

If the effects of the viscosity are not important, the expression (2.10)—derived from (2.2)—is valid.

$$\begin{cases} \dfrac{\partial u}{\partial t} + u\dfrac{\partial u}{\partial x} + v\dfrac{\partial u}{\partial y} + w\dfrac{\partial u}{\partial z} - fv = -\dfrac{1}{\rho}\dfrac{\partial p}{\partial x} \\ \dfrac{\partial v}{\partial t} + u\dfrac{\partial v}{\partial x} + v\dfrac{\partial v}{\partial y} + w\dfrac{\partial v}{\partial z} + fu = -\dfrac{1}{\rho}\dfrac{\partial p}{\partial y} \\ \dfrac{\partial w}{\partial t} + u\dfrac{\partial w}{\partial x} + v\dfrac{\partial w}{\partial y} + w\dfrac{\partial w}{\partial z} = -\dfrac{1}{\rho}\dfrac{\partial p}{\partial z} - g \end{cases} \tag{2.10}$$

If the pressure is expressed as $p = -\rho g z + \tilde{p}$, the system of equations (2.10) is rewritten as (2.11).

$$\begin{cases} \dfrac{\partial u}{\partial t} + u\dfrac{\partial u}{\partial x} + v\dfrac{\partial u}{\partial y} + w\dfrac{\partial u}{\partial z} - fv = -\dfrac{1}{\rho}\dfrac{\partial \widetilde{p}}{\partial x} \\ \dfrac{\partial v}{\partial t} + u\dfrac{\partial v}{\partial x} + v\dfrac{\partial v}{\partial y} + w\dfrac{\partial v}{\partial z} + fu = -\dfrac{1}{\rho}\dfrac{\partial \widetilde{p}}{\partial y} \\ \dfrac{\partial w}{\partial t} + u\dfrac{\partial w}{\partial x} + v\dfrac{\partial w}{\partial y} + w\dfrac{\partial w}{\partial z} = -\dfrac{1}{\rho}\dfrac{\partial \widetilde{p}}{\partial z} \end{cases} \tag{2.11}$$

The importance of each of the terms can be determined by analysing the orders of magnitude of the terms through (2.12).

$$\begin{cases} \dfrac{U}{T} + \dfrac{U^2}{L} + \dfrac{U^2}{L} + \dfrac{UW}{D} - fU \sim -\dfrac{1}{\rho}\dfrac{\tilde{P}}{L} \\ \dfrac{U}{T} + \dfrac{U^2}{L} + \dfrac{U^2}{L} + \dfrac{UW}{D} + fU \sim -\dfrac{1}{\rho}\dfrac{\tilde{P}}{L} \\ \dfrac{W}{T} + \dfrac{UW}{L} + \dfrac{UW}{L} + \dfrac{W^2}{D} \sim -\dfrac{1}{\rho}\dfrac{\tilde{P}}{D} \end{cases} \tag{2.12}$$

In order to ensure consistency in the equations concerning u and v, the pressure gradients must be of the same order of the accelerations. Therefore, one can write (2.13). Note that $U/T = U^2/L = UW/D$ and $W/T = UW/L = W^2/D$.

$$\tilde{P} \sim \rho L \max\left(\left\{\frac{U^2}{L}, fU\right\}\right) \tag{2.13}$$

Note that $\tilde{P}$ is obtained for a distance L on the horizontal scale, but this does not necessarily has to be the same variation needed in the equation for w in order to have consistent accelerations. The ratio between the vertical acceleration and the vertical pressure gradient on $D \ll L$ is written as (2.14).

$$\rho\frac{\frac{dw}{dt}}{\frac{\partial p}{\partial \tilde{z}}} \sim \rho\frac{\frac{WU}{L}}{\frac{\tilde{P}}{D}} = \delta^2\frac{\frac{U}{L}}{\max\left(\left\{\frac{U}{L}, f\right\}\right)} = \delta^2 \max\left(\{1, \mathtt{Ro}\}\right) \tag{2.14}$$

Assuming $\delta^2\mathtt{Ro} < \delta^2$, it is seen that the ratio referenced above cannot be greater than δ^2. From this result it can be stated that the vertical dynamic pressure gradient $\partial\tilde{p}/\partial z$ must be δ^2 smaller than the horizontal pressure gradient, and so the vertical acceleration dw/dt. Expression (2.15) is true.

$$\frac{\partial p}{\partial z} = -\rho g + O\left(\delta^2\right) = -\rho g + \cancelto{\approx 0}{\frac{\partial \tilde{p}}{\partial z}} \simeq -\rho g \tag{2.15}$$

Integrating the expression (2.15) one can obtain (2.16).

$$p = -\rho g z + A\left(x, y, t\right) \tag{2.16}$$

If the pressure on the surface $z = D + \eta$ is constant and equal to p_0, the value of $A = A\left(x, y, t\right)$ is found to be (2.17).

$$\begin{aligned} p\left(x, y, z, t\right) &= p_0 \\ -\rho g h + A\left(x, y, t\right) &= p_0 \\ A\left(x, y, t\right) &= p_0 + \rho g\left(D + \eta\right) \end{aligned} \tag{2.17}$$

The result above can be used to derive the expression (2.18), which states that the pressure is simply the weight of the liquid column above the surface.

$$p = p\,(x, y, z, t) = -\rho g z + \underbrace{p_0 + \rho g D + \rho g \eta}_{\tilde{p}} \tag{2.18}$$

Because $z = D + \eta$—where η is the perturbation of the free surface—the expression (2.19) is also valid. See Figure 2.3 for a visual representation of the variables.

$$\begin{cases} \dfrac{\partial \tilde{p}}{\partial x} = \rho g \dfrac{\partial \eta}{\partial x} \\ \dfrac{\partial \tilde{p}}{\partial y} = \rho g \dfrac{\partial \eta}{\partial y} \\ \dfrac{\partial \tilde{p}}{\partial z} = 0 \end{cases} \tag{2.19}$$

Because $h = h\,(x, y, t)$, the first to expressions in (2.10) can be rewritten to (2.20), where w has been neglected before u and v.

$$\begin{cases} \dfrac{\partial u}{\partial t} + u\dfrac{\partial u}{\partial x} + v\dfrac{\partial u}{\partial y} - fv = -g\dfrac{\partial \eta}{\partial x} \\ \dfrac{\partial v}{\partial t} + u\dfrac{\partial v}{\partial x} + v\dfrac{\partial v}{\partial y} + fu = -g\dfrac{\partial \eta}{\partial y} \end{cases} \tag{2.20}$$

In (2.20) it is shown that u and v do not depend on z.

Returning to the continuity equation with $w = w(x, y, z, t)$, the equation (2.21) is obtained after noting that η is not a function of z from $\nabla \cdot \vec{u} = 0$.

$$\begin{aligned} \frac{\partial w}{\partial z} &= -\frac{\partial u}{\partial x} - \frac{\partial v}{\partial y} \\ w\,(x, y, z, t) &= -\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) z + B\,(x, y, t) \end{aligned} \tag{2.21}$$

For $z = h_B$, the following can be derived so as to obtain $B(x, y, t)$ in (2.22).

$$\begin{aligned} w = \frac{dz}{dt} &= \cancelto{0}{\frac{\partial h_B}{\partial t}} + u\frac{\partial h_B}{\partial x} + v\frac{\partial h_B}{\partial y} \\ &= u\frac{\partial h_B}{\partial x} + v\frac{\partial h_B}{\partial y} \\ -\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) h_B + B\,(x, y, t) &= u\frac{\partial h_B}{\partial x} + v\frac{\partial h_B}{\partial y} \\ B\,(x, y, t) &= u\frac{\partial h_B}{\partial x} + v\frac{\partial h_B}{\partial y} + \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) h_B \end{aligned} \tag{2.22}$$

With this result, it can be demonstrated that w is a function linearly dependent on z as in (2.23).

$$\begin{aligned} w\,(x, y, z, t) &= -\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) z + u\frac{\partial h_B}{\partial x} + v\frac{\partial h_B}{\partial y} + \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) h_B \\ &= \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)(h_B - z) + u\frac{\partial h_B}{\partial x} + v\frac{\partial h_B}{\partial y} \end{aligned} \tag{2.23}$$

By using (2.23) and the derivative of $z = D + \eta\,(x, y, t)$ as in (2.24), (2.25) is found.

$$w\,(x, y, z, t) = \frac{d\eta}{dt} = \frac{\partial \eta}{\partial t} + u\frac{\partial \eta}{\partial x} + v\frac{\partial \eta}{\partial y} \tag{2.24}$$

$$\left(\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}\right)(h_B-D-\eta)+u\frac{\partial h_B}{\partial x}+v\frac{\partial h_B}{\partial y}=\frac{\partial \eta}{\partial t}+u\frac{\partial \eta}{\partial x}+v\frac{\partial \eta}{\partial y}$$

$$\frac{\partial \eta}{\partial t}+u\frac{\partial}{\partial x}(\eta-h_B)+v\frac{\partial}{\partial y}(\eta-h_B)-\left(\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}\right)(h_B-\eta-D)=0$$

$$\frac{\partial \eta}{\partial t}+u\frac{\partial}{\partial x}(\eta+D-h_B)+v\frac{\partial}{\partial y}(\eta+D-h_B)-\left(\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}\right)(h_B-\eta-D)=0$$

$$\frac{\partial \eta}{\partial t}+\frac{\partial}{\partial x}\left[u(\eta+D-h_B)\right]+\frac{\partial}{\partial y}\left[v(\eta+D-h_B)\right]=0 \tag{2.25}$$

From (2.25) and with $h=D+\eta-h_B$ the following in (2.26) is demonstrated.

$$\frac{\partial \eta}{\partial t}+\nabla\cdot(h\vec{u})=0 \tag{2.26}$$

Because $h_B=h_B(x,y)$, then (2.27) holds also true.

$$\frac{\partial h}{\partial t}+\nabla\cdot(h\vec{u})=0 \tag{2.27}$$

The shallow water model can be expressed as the set of equations(2.28) constituted by the expressions in (2.20) and in (2.27).

The shallow water equations

$$\begin{cases}\dfrac{\partial u}{\partial t}+u\dfrac{\partial u}{\partial x}+v\dfrac{\partial u}{\partial y}-fv=-g\dfrac{\partial \eta}{\partial x}\\ \dfrac{\partial v}{\partial t}+u\dfrac{\partial v}{\partial x}+v\dfrac{\partial v}{\partial y}+fu=-g\dfrac{\partial \eta}{\partial y}\\ \dfrac{\partial h}{\partial t}+\nabla\cdot(h\vec{u})=0\\ h-D-\eta+h_B=0\end{cases} \tag{2.28}$$

2.2.3 Conservation of the potential vorticity

Vorticity is written as $\vec{\omega}=\nabla\times\vec{u}$ and is a measure of the tendency of the flow to rotate around a point. The expression can be rewritten in Cartesian coordinates as shown in (2.29). Note that $u=u(x,y)$ and $v=v(x,y)$.

$$\begin{cases}\omega_x=\dfrac{\partial w}{\partial y}-\dfrac{\partial v}{\partial z}=\dfrac{\partial w}{\partial y}\\ \omega_y=\dfrac{\partial u}{\partial z}-\dfrac{\partial w}{\partial x}=-\dfrac{\partial w}{\partial x}\\ \omega_z=\dfrac{\partial v}{\partial x}-\dfrac{\partial u}{\partial y}=\dfrac{\partial v}{\partial x}-\dfrac{\partial u}{\partial y}\end{cases} \tag{2.29}$$

The order of magnitude of terms in (2.29) are presented in (2.30).

$$\begin{cases}O(\omega_x)\sim\dfrac{W}{L}\sim\delta\dfrac{U}{L}\\ O\left(\omega_y\right)\sim\dfrac{W}{L}\sim\delta\dfrac{U}{L}\\ O(\omega_z)\sim\dfrac{U}{L}\end{cases} \tag{2.30}$$

From (2.30) it can be seen that ω is mostly determined by the ω_z component—that is the vertical component—so (2.31) is true.

$$\vec{\omega} \approx \omega_z \boldsymbol{k} = \varsigma \boldsymbol{k} \tag{2.31}$$

A system of equations like (2.32) can be built with (2.20) and (2.31) in order to obtain the desired result.

$$\begin{cases} \dfrac{\partial u}{\partial t} + u\dfrac{\partial u}{\partial x} + v\dfrac{\partial u}{\partial y} - fv = -g\dfrac{\partial h}{\partial x} \\ \dfrac{\partial v}{\partial t} + u\dfrac{\partial v}{\partial x} + v\dfrac{\partial v}{\partial y} + fu = -g\dfrac{\partial h}{\partial y} \\ \varsigma = \dfrac{\partial v}{\partial x} - \dfrac{\partial u}{\partial y} \end{cases} \tag{2.32}$$

By manipulating the first two lines in (2.32) as it is shown in (2.33) and subtracting one from the other, (2.34) is obtained

$$\begin{cases} \dfrac{\partial^2 u}{\partial t \partial y} + \dfrac{\partial^2 u}{\partial x \partial y} + u\dfrac{\partial^2 u}{\partial x \partial y} + \dfrac{\partial v}{\partial y} \cdot \dfrac{\partial u}{\partial y} + v\dfrac{\partial^2 u}{\partial y^2} - f\dfrac{\partial v}{\partial y} = -g\dfrac{\partial^2 h}{\partial x \partial y} \\ \dfrac{\partial^2 v}{\partial t \partial x} + \dfrac{\partial u}{\partial x} \cdot \dfrac{\partial v}{\partial x} + u\dfrac{\partial^2 v}{\partial x^2} + \dfrac{\partial^2 v}{\partial x \partial y} + v\dfrac{\partial^2 v}{\partial x \partial y} + f\dfrac{\partial u}{\partial x} = -g\dfrac{\partial^2 h}{\partial x \partial y} \end{cases} \tag{2.33}$$

$$\begin{aligned} \frac{\partial^2 u}{\partial t \partial y} + \frac{\partial^2 u}{\partial x \partial y} + u\frac{\partial^2 u}{\partial x \partial y} + \frac{\partial v}{\partial y} \cdot \frac{\partial u}{\partial y} + v\frac{\partial^2 u}{\partial y^2} - f\frac{\partial v}{\partial y} \\ - \frac{\partial^2 v}{\partial t \partial x} - \frac{\partial u}{\partial x} \cdot \frac{\partial v}{\partial x} - u\frac{\partial^2 v}{\partial x^2} - \frac{\partial^2 v}{\partial x \partial y} - v\frac{\partial^2 v}{\partial x \partial y} - f\frac{\partial u}{\partial x} = 0 \end{aligned} \tag{2.34}$$

The last result can be rearranged accordingly as in (2.35), and with the third line of (2.32), the expression (2.36) is obtained.

$$\begin{aligned} \frac{\partial}{\partial t}\left(\frac{\partial u}{\partial y} - \frac{\partial v}{\partial x}\right) + u\left(\frac{\partial^2 u}{\partial x \partial y} - \frac{\partial^2 v}{\partial x^2}\right) + v\left(\frac{\partial^2 u}{\partial y^2} - \frac{\partial^2 v}{\partial x \partial y}\right) \\ - f\left(\frac{\partial v}{\partial y} + \frac{\partial u}{\partial x}\right) + \left(\frac{\partial u}{\partial y} - \frac{\partial v}{\partial x}\right)\left(\frac{\partial v}{\partial y} + \frac{\partial u}{\partial x}\right) = 0 \end{aligned} \tag{2.35}$$

Using the expression (2.27) on (2.36), the following in (2.37) is derived. Note that f is constant so as to find the expression (2.38).

$$\begin{aligned} -\frac{\partial \varsigma}{\partial t} - u\frac{\partial \varsigma}{\partial x} - v\frac{\partial \varsigma}{\partial y} - f\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) - \varsigma\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) &= 0 \\ \frac{\partial \varsigma}{\partial t} + u\frac{\partial \varsigma}{\partial x} + v\frac{\partial \varsigma}{\partial y} + (f + \varsigma)\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) &= 0 \qquad (2.36) \\ \frac{\partial \varsigma}{\partial t} + \nabla \cdot (\varsigma \vec{u}) + (f + \varsigma)\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) &= 0 \\ \frac{\partial \varsigma}{\partial t} + \nabla \cdot (\varsigma \vec{u}) - (f + \varsigma)\frac{1}{h}\frac{\partial h}{\partial t} &= 0 \qquad (2.37) \\ \frac{d\varsigma}{dt} - (f + \varsigma)\frac{1}{h}\frac{\partial h}{\partial t} &= 0 \\ \frac{d(f + \varsigma)}{dt} - (f + \varsigma)\frac{1}{h}\frac{\partial h}{\partial t} &= 0 \\ \frac{1}{h}\frac{d(f + \varsigma)}{dt} - (f + \varsigma)\frac{1}{h^2}\frac{\partial h}{\partial t} &= 0 \\ \frac{d}{dt}\left(\frac{f + \varsigma}{h}\right) &= 0 \qquad (2.38) \end{aligned}$$

Potential vorticity

The result in (2.38) indicates that the potential vorticity

$$\Pi_s = \frac{\varsigma + f}{h} \tag{2.39}$$

is conserved following the fluid parcel, and therefore can be used as a passive tracer similar to a cloud.

2.3 The shallow water model in other reference systems

The shallow water equations expressed in the Cartesian coordinate system, the ones presented in (2.28), can be used to model local phenomena together with the f-plane and β-plane approximations. These do not use (2.8) but approximations of the expression instead. For the f-plane, the Coriolis parameter is set to be constant while for the β-plane, it is set to vary linearly with respect to the latitude.

The reader will find it intuitive to use a spherical coordinates instead of Cartesian coordinates to model large-scale phenomena as the planets are ball-like. The use of a spherical coordinate system leads to more accurate results, but a generalization of the spherical coordinate system, the ellipsoidal coordinate system, suits much better. Most of the planets are spheroid-like[1] with polar radii being shorter than equatorial radii, resembling a ball flattened at the poles[2]. The Gas Giants of the Solar System are known for their notable flattening, with Jupiter having a flattening of around $1/16$ and Saturn of $\approx 1/10$. The use of "spheroidal" coordinates make sense when large regions of the planet's atmosphere want to be modelled.

In fact, simulations of giant planet's atmospheres[3;4] have used this coordinate system. For example, the original–legacy Shallow Worlds, written by Enrique García, implements this coordinate system (see (2.40)). These equations can be derived through tensor calculus.

The shallow water equations written in the ellipsoidal coordinate system

For a longitude λ and a latitude φ,

$$\begin{cases} \dfrac{\partial u}{\partial t} + \dfrac{u}{r_Z(\varphi)}\dfrac{\partial u}{\partial \lambda} + \dfrac{v}{r_M(\varphi)}\dfrac{\partial u}{\partial \varphi} - \left(2\Omega \sin\varphi + \dfrac{\sin\varphi}{r_Z(\varphi)}u\right) v = -\dfrac{g}{r_Z(\varphi)}\dfrac{\partial \eta}{\partial x} \\ \dfrac{\partial v}{\partial t} + \dfrac{u}{r_Z(\varphi)}\dfrac{\partial v}{\partial \lambda} + \dfrac{v}{r_M(\varphi)}\dfrac{\partial v}{\partial \varphi} + \left(2\Omega \sin\varphi + \dfrac{\sin\varphi}{r_Z(\varphi)}u\right) u = -\dfrac{g}{r_M(\varphi)}\dfrac{\partial \eta}{\partial y} \\ \dfrac{\partial h}{\partial t} + \dfrac{1}{r_Z(\varphi)}\dfrac{\partial (hu)}{\partial \lambda} + \dfrac{1}{r_M(\varphi)}\dfrac{\partial (hv)}{\partial \varphi} - \dfrac{\sin\varphi}{r_Z(\varphi)}hv = 0 \\ h - D - \eta + h_B = 0 \end{cases} \tag{2.40}$$

with

$$r_Z(\varphi) = \frac{R_E R_P}{\sqrt{1 + \varepsilon^{-2}\tan^2\varphi}}$$

$$r_M(\varphi) = \frac{R_E}{\varepsilon^2}\left(\frac{r_Z(\varphi)}{R_E \cos\varphi}\right)^3$$

It should be noted that the latitude φ referenced in the equations above is the geodetic latitude, which will be identified as planetographic latitude φ_g from now on. Longitude will be only identified as λ hereinafter, as the cross sections of the planet cutting in planes parallel to the equator are circumferences. A cut of a spheroid with the different variables indicated is presented in Figure 2.4.

[1] Spheroid is understood as an ellipsoid of revolution.
[2] The technical term for this surface is *oblate spheroid*.

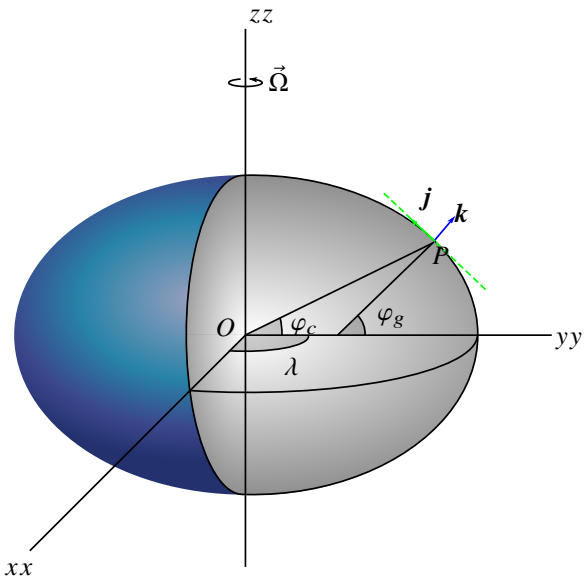


Figure 2.4: Reference system for a spheroid

An experienced programmer will see that (2.28) and (2.40) share the same terms, but in the latter, geometrical factors appear. In the field of numerical solvers, this can be translated into having the same core both for a plane and for an ellipsoid. An example of the core of a solver is presented in chapter 5.

2.4 Boundary conditions in atmosphere simulations

For the study of storms and other phenomena, the boundary condition (BC) typically used are those of a channel, as presented in Figure 2.5. The boundaries parallel to the equator are walls, which have to be far from the storm to be studied to avoid unrealistic interactions. These interactions are reduced if zonal winds[3] are present, as they restrict the movement of the storm in the y direction.

The interaction between the storm and its wake can be mimicked if a periodic BC is imposed in the x direction. In most simulations, domains usually do not need to encompass up to $\Delta\lambda = 360°$, and simulations of domains with a third of this range may be realistic enough for individual storms because, generally in wide enough domains, the wake exits the domain stably enough. In figure Figure 2.5, observe how the incoming stream is almost the same as the flow that exits the domain.

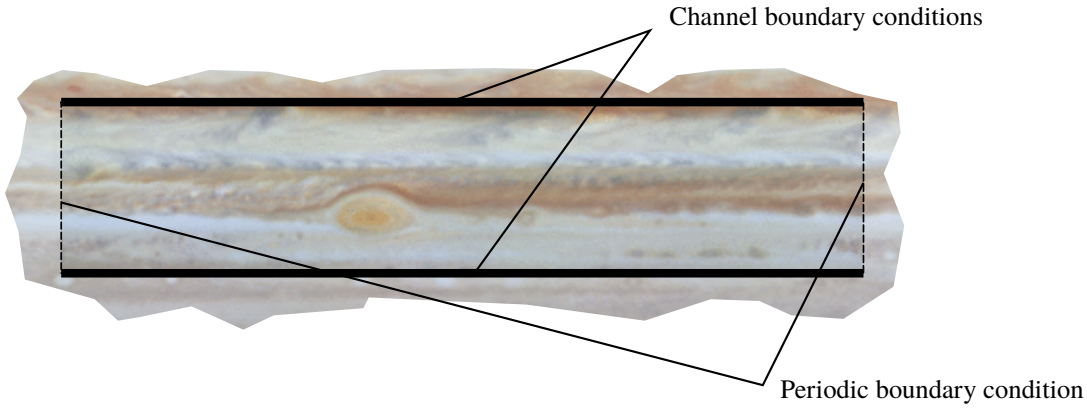


Figure 2.5: Boundary conditions for the study of planet storms (Credit to: NASA/JPL/University of Arizona)

Further information on how these conditions have been achieved in Shallow Worlds is found in section 3.7.

[3]Zonal wind is air moving along a given parallel. Information on zonal winds can be found in the work of Arnau Sabatés.

2.5 Notes on the shallow water model

The model can only be applied in the regions of the atmosphere where the viscosity is negligible. The planet's boundary layer cannot be simulated as friction forces are too important in this region. Further explanation of this can be found in the work of Arnau Sabatés[8].

The shallow water model is thoroughly used in a variety of other fields. The equations are also used [as presented] in the prediction of buoy trajectories and ocean currents due to the similarities with the atmosphere. Note that the depth of the oceans is practically negligible in comparison to a reference length, similarly to the thickness of the atmosphere compared to the planet's reference radius. Other applications include hydraulic engineering, where these equations are widely used together with other approximations such as the Boussinesq approximation. In situations where the fluid layer is heavily stratified, multilayer shallow water models can be used to take into account factors that vary with height.

The shallow water model can be computationally solved rapidly, which is very attractive towards predicting large scale phenomena with a limited infrastructure. Details on the implementation of the model in a solver is found below.

CHAPTER 3

Numerical integration of the shallow water equations

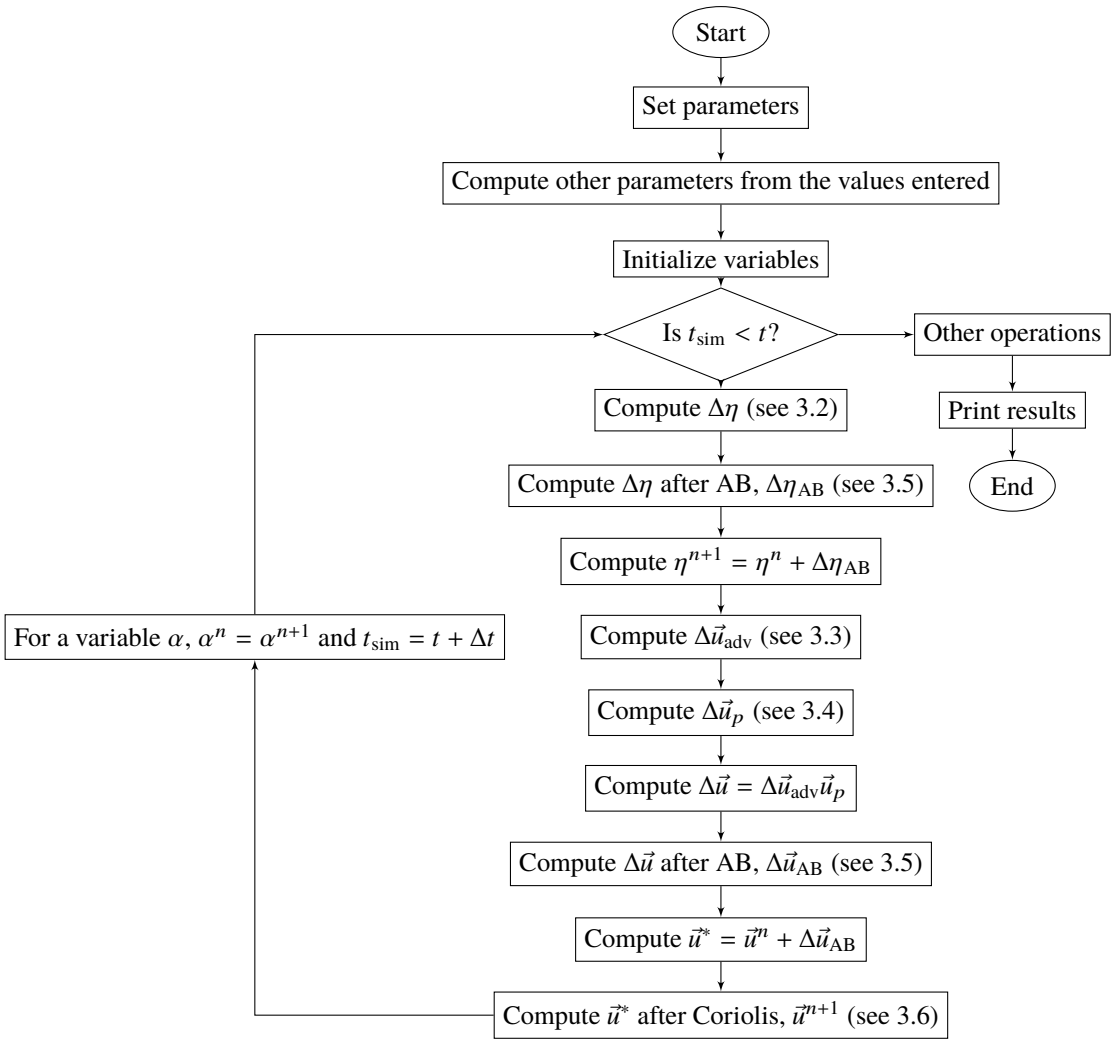


Figure 3.1: Solver flowchart

The shallow water equations can be integrated by means of numerical methods. Due to the nature of the equations, these are solved by slowly reaching the solution by computing the solution of smaller time ranges that take the previous result as initial conditions. In fact, a loop is carried out while the simulation time is not larger than the end time. The structure of the solver can be seen in Figure 3.1.

The shallow water equations are not integrated explicitly, but by adding the contributions of the different terms to the temporal change of u, v and h according to the rules of Partial Differential Equation (PDE) integration. For example, in order to solve the conservation of momentum equation—which contains the advection term, the gradient of pressure, the Coriolis force and geometrical parameters—the contributions of the advection and pressure to $\vec{u}$ have to be added *before* the Coriolis te is taken into account

Numerical integration of Ordinary Differential Equation (ODE)s and PDEs can be achieved by a number of different methods. Those used at the core of Shallow Worlds have been justified in the work of Arnau Sabatés[8].

A good understanding of the methodology is needed to create a solver like Shallow Worlds. In fact, the design of th solver has been heavily conditioned by the variables needed in the different functions that solve a time step.

3.1 Discretization of the shallow water equations

The solver performs the operations on discretized variables, i.e. the variables are manipulated as collections of points instead of continua. Because high precision results are desired, at least second order approximations have to be achieved. In traditional, less complex but more intuitive methods, higher order approximations are achieved by increasing the resolution of the grid, which increases the number of control volume (CV)s. There are other methods that achieve higher order approximations by referencing the different equations to specific points of the domain so the required number of nodes is not increased with respect to methods that achieve first-order approximations. These methods use the so-called *staggered* grids and they are often used in fluid dynamics[2].

With the use of an Arakawa C-grid, second-second order approximations can be achieved by having the same number of points for every variable and thereby constituting matrices of the same size. This facilitates memory management as memory allocation may be done with the same expressions for each of the equal-size matrices. Despite the advantages, an important drawback is that the different variables are referenced in the points shown in Figure 3.2, adding complexity in the programming phase. In the referenced figure, the semitransparent regions group the different variables, so

- the grey area contains centred nodes,
- the red area contains the points staggered in the x direction, and
- the blue area contains the points staggered in the y direction.

If the layers are understood as matrices, all of them are of size n_x-by-n_y, as seen before.

The non coloured area surrounding the domain of length L is called the *halo* and will be introduced in chapter 3. The Finite Difference Method (FDM) and the Total Variation Diminishing (TVD) schemes require having this region in order to perform the approximations.

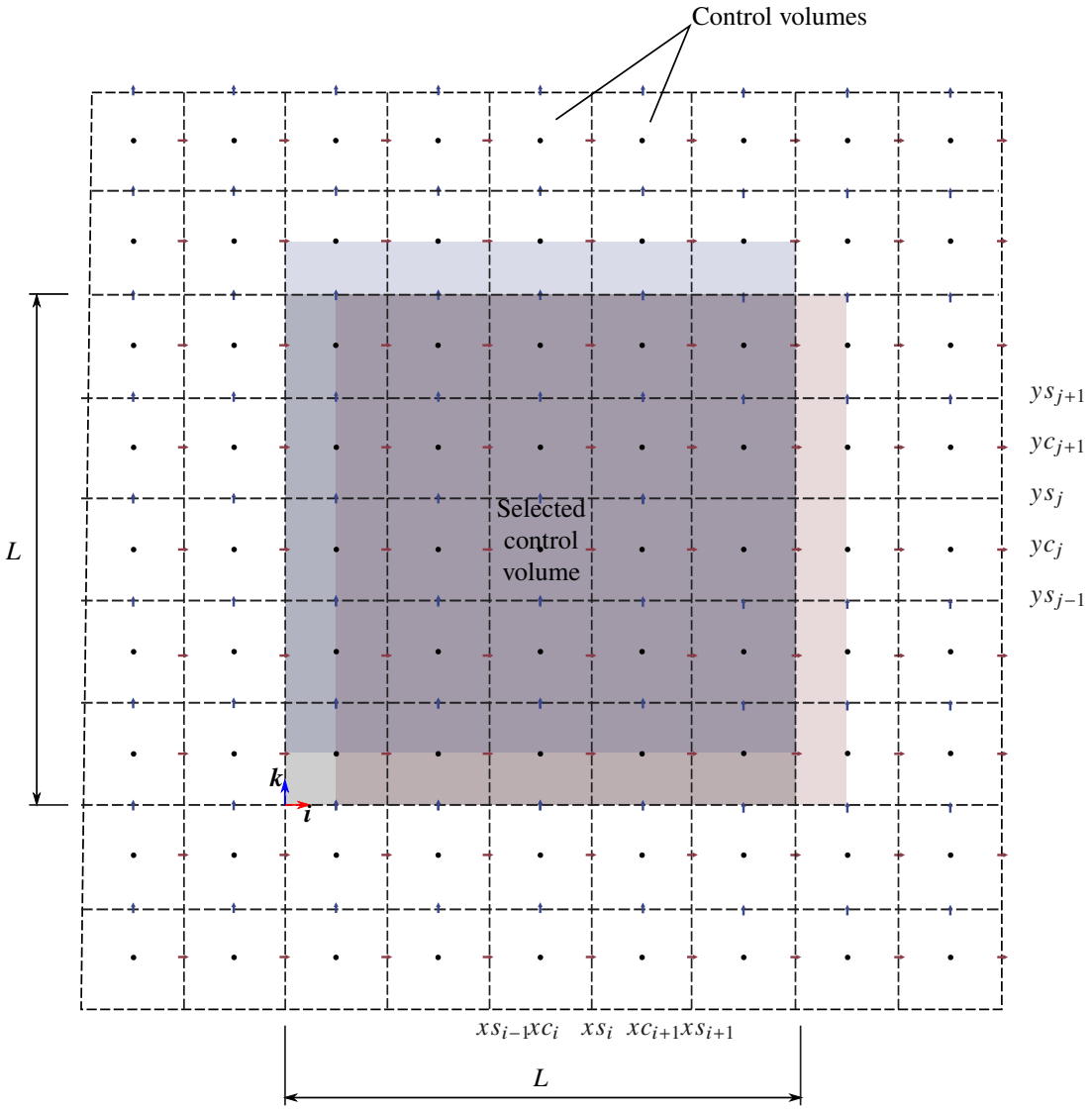


Figure 3.2: Domain with a staggered-grid

For a given CV of the domain, the variables are arranged as in Figure 3.3, where the disposition of the variables can be clearly seen. The pressure is found at the centre of the square cell, while the horizontal velocity is computed at the right side and the vertical velocity, at the top side.

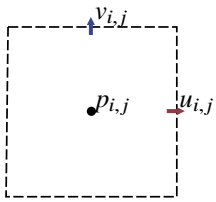


Figure 3.3: Staggered grid cell

3.2 Integration of the continuity equation

The continuity equation

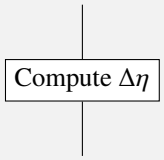


Figure 3.4: η integration block

The continuity equation is written as

$$\frac{\partial h}{\partial t} + \nabla \cdot (h\vec{u}) = 0 \tag{3.1}$$

and can be integrated in Shallow Worlds with `solve_cons_h(...)`, which is the block in Figure 3.4 of the flowchart in Figure 3.1.

The integration of the h from equation (3.1) may be carried out using the TVD Superbee numeric scheme[6]. Because the h is positioned at the centred coordinates as shown in Figure 3.5 the equation is discretized as follows in (3.2).

As seen in (2.28), h and η are related through h_B. Because the integration of the pressure term (see section 3.4) is written in terms of η, Shallow Worlds stores the η field—instead of h—in memory. The values of the h field can be accessed by means of `H(...)` and while this macro cannot be used to set the value of h at a point i, j of the domain, the way it is defined is useful enough for the majority of applications.

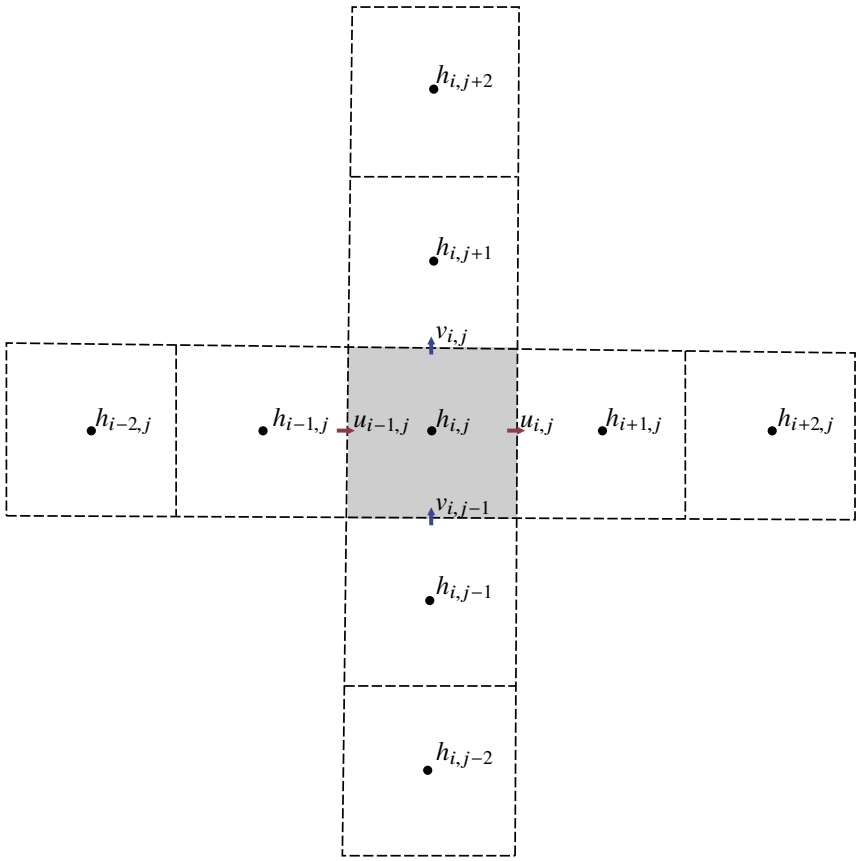


Figure 3.5: Staggered grid cell and its halo

$$\frac{\partial h}{\partial t} = -\frac{\partial (uh)}{\partial x} - \frac{\partial (vh)}{\partial y} \tag{3.2}$$

$$dh = -\frac{\partial (uh)}{\partial x}dt - \frac{\partial (vh)}{\partial y}dt$$

$$\Delta h \approx -\frac{\partial (uh)}{\partial x}\Delta t - \frac{\partial (vh)}{\partial y}\Delta t$$

$$\Delta h \approx C_w h_w - C_e h_e + C_s h_s - C_n h_n \tag{3.3}$$

The Courant numbers appearing in (3.3) are computed as show in (3.4).

$$\begin{aligned} C_w &= u^n_{i-1,j}\frac{\Delta t}{\Delta x} \\ C_e &= u^n_{i,j}\frac{\Delta t}{\Delta x} \\ C_s &= v^n_{i,j-1}\frac{\Delta t}{\Delta y} \\ C_n &= v^n_{i,j}\frac{\Delta t}{\Delta y} \end{aligned} \tag{3.4}$$

Using the expressions in (3.5) for u and v one can obtain C^+_w, C^-_w, C^+_e, C^-_e, C^+_n, C^-_n, C^+_s and C^-_s, which appear in expression (3.6).

$$\begin{aligned} u^+ &= \tfrac{u+|u|}{2} \quad v^+ = \tfrac{v+|v|}{2} \\ u^- &= \tfrac{u-|u|}{2} \quad v^+ = \tfrac{v-|v|}{2} \end{aligned} \tag{3.5}$$

$$\begin{aligned} \Delta h &\approx C_w h_w - C_e h_e + C_s h_s - C_n h_n \\ &\approx C^+_w h^+_w + C^-_w h^-_w - C^+_e h^+_e - C^-_e h^-_e + C^+_s h^+_s + C^-_s h^-_s - C^+_n h^+_n - C^-_n h^-_n \end{aligned} \tag{3.6}$$

Combining expressions (3.4) and (3.5), the expressions in (3.8) are found and are used in (3.6) together with h^+_w, h^-_w, h^+_e, h^-_e, h^+_n, h^-_n, h^+_s and h^-_s, which are computed as in (3.7). From the last expression it can be seen that the solver needs to access the values of the surrounding cells in order to perform the operation in the selected CV. It must read the two immediate values per side of the CV, which makes the cell required to perform the operations to be like the one presented in Figure 3.5. The easiest way to fit this cell in every position of the domain is to ensure that the discretized domain is surrounded by a halo that adds two CVs per side of the domain and position, as shown in Figure 3.2. In other words, the matrices that contain the values of the variables need to have two extra sets of rows and columns to accommodate the points that fall on the halo.

$$\begin{aligned}
h^+_w &= h^n_{i-1,j} + \tfrac{1}{2}\Psi\left(r^+_{i-1,j}\right)\left(1 - C^+_w\right)\left(h^n_{i,j} - h^n_{i-1,j}\right) && \text{with } r^+_{i-1,j} = \tfrac{h^n_{i-1,j} - h^n_{i-2,j}}{h^n_{i,j} - h^n_{i-1,j}} \\
h^-_w &= h^n_{i,j} - \tfrac{1}{2}\Psi\left(r^-_{i-1,j}\right)\left(1 + C^-_w\right)\left(h^n_{i,j} - h^n_{i-1,j}\right) && \text{with } r^-_{i-1,j} = \tfrac{h^n_{i+1,j} - h^n_{i,j}}{h^n_{i,j} - h^n_{i-1,j}} \\
h^+_e &= h^n_{i,j} + \tfrac{1}{2}\Psi\left(r^+_{i,j}\right)\left(1 - C^+_e\right)\left(h^n_{i+1,j} - h^n_{i,j}\right) && \text{with } r^+_{i,j} = \tfrac{h^n_{i,j} - h^n_{i-1,j}}{h^n_{i+1,j} - h^n_{i,j}} \\
h^-_e &= h^n_{i+1,j} - \tfrac{1}{2}\Psi\left(r^-_{i,j}\right)\left(1 + C^-_e\right)\left(h^n_{i+1,j} - h^n_{i,j}\right) && \text{with } r^-_{i,j} = \tfrac{h^n_{i+2,j} - h^n_{i+1,j}}{h^n_{i+1,j} - h^n_{i,j}} \\
h^+_s &= h^n_{i,j-1} + \tfrac{1}{2}\Psi\left(r^+_{i,j-1}\right)\left(1 - C^+_s\right)\left(h^n_{i,j} - h^n_{i,j-1}\right) && \text{with } r^+_{i,j-1} = \tfrac{h^n_{i,j-1} - h^n_{i,j-2}}{h^n_{i,j} - h^n_{i,j-1}} \\
h^-_s &= h^n_{i,j} - \tfrac{1}{2}\Psi\left(r^-_{i,j-1}\right)\left(1 + C^-_s\right)\left(h^n_{i,j} - h^n_{i,j-1}\right) && \text{with } r^-_{i,j-1} = \tfrac{h^n_{i,j+1} - h^n_{i,j}}{h^n_{i,j} - h^n_{i,j-1}} \\
h^+_n &= h^n_{i,j} + \tfrac{1}{2}\Psi\left(r^+_{i,j}\right)\left(1 - C^+_n\right)\left(h^n_{i,j+1} - h^n_{i,j}\right) && \text{with } r^+_{i,j} = \tfrac{h^n_{i,j} - h^n_{i,j-1}}{h^n_{i,j+1} - h^n_{i,j}} \\
h^-_n &= h^n_{i,j+1} - \tfrac{1}{2}\Psi\left(r^-_{i,j}\right)\left(1 + C^-_n\right)\left(h^n_{i,j+1} - h^n_{i,j}\right) && \text{with } r^-_{i,j} = \tfrac{h^n_{i,j+2} - h^n_{i,j+1}}{h^n_{i,j+1} - h^n_{i,j}}
\end{aligned} \tag{3.7}$$

$$
\begin{aligned}
C_{w^+} &= \frac{u_{i-1,j} + \left|u_{i-1,j}\right|}{2} \cdot \frac{\Delta t}{\Delta x} \\
C_{w^-} &= \frac{u_{i-1,j} - \left|u_{i-1,j}\right|}{2} \cdot \frac{\Delta t}{\Delta x} \\
C_{e^+} &= \frac{u_{i,j} + \left|u_{i,j}\right|}{2} \cdot \frac{\Delta t}{\Delta x} \\
C_{e^-} &= \frac{u_{i,j} - \left|u_{i,j}\right|}{2} \cdot \frac{\Delta t}{\Delta x} \\
C_{s^+} &= \frac{v_{i,j-1} + \left|v_{i,j-1}\right|}{2} \cdot \frac{\Delta t}{\Delta y} \\
C_{s^-} &= \frac{v_{i,j-1} - \left|v_{i,j-1}\right|}{2} \cdot \frac{\Delta t}{\Delta y} \\
C_{n^+} &= \frac{v_{i,j} + \left|v_{i,j}\right|}{2} \cdot \frac{\Delta t}{\Delta y} \\
C_{n^-} &= \frac{v_{i,j} - \left|v_{i,j}\right|}{2} \cdot \frac{\Delta t}{\Delta y}
\end{aligned}
\tag{3.8}
$$

When the continuity equation is expressed in an ellipsoidal coordinate system, a geometric term appears, as seen in (2.40). This term should be directly added to Δh in (3.6) as seen in the core of Shallow Worlds (see `do_core_loop(...)`). The function that computes Δh is `solve_cons_h(...)`.

3.3 Integration of the advection term

The temporal change of the velocity due to advection

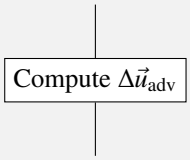


Figure 3.6: $\vec{u}_{\text{adv}}$ integration block

A system where the only cause of temporal change of $\vec{u}$ is the advection, is characterized as

$$
\frac{d\vec{u}}{dt} + \vec{u} \cdot \nabla\vec{u} = 0 \tag{3.9}
$$

or, in Cartesian coordinates, as

$$
\begin{cases}
\dfrac{\partial u}{\partial t} + u\dfrac{\partial u}{\partial x} + v\dfrac{\partial u}{\partial y} = 0 \\
\dfrac{\partial v}{\partial t} + u\dfrac{\partial v}{\partial x} + v\dfrac{\partial v}{\partial y} = 0
\end{cases}
\tag{3.10}
$$

In order to integrate the equation of the conservation of momentum, the Shallow Worlds solver computes the contribution of the advection to the velocity $\vec{v}$ by calling `solve_adv_u(...)` and `solve_adv_v(...)`. These functions are part of the block in Figure 3.6 of the flowchart in Figure 3.1.

The equations in (3.10) above can be discretized as follows in (3.11). The following is the scheme on which the routines for the x direction ($B = u$) and the y direction ($B = v$) are based.

$$\frac{\partial B}{\partial t} = -u\frac{\partial B}{\partial x} - v\frac{\partial B}{\partial y} \tag{3.11}$$

$$\frac{\partial B}{\partial t} = -\frac{\partial (uB)}{\partial x} - \frac{\partial (vB)}{\partial y} + B\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)$$

$$dB = -\frac{\partial (uB)}{\partial x}dt - \frac{\partial (vB)}{\partial y}dt + B\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)dt$$

$$\Delta B \approx -\frac{\partial (uB)}{\partial x}\Delta t - \frac{\partial (vB)}{\partial y}\Delta t + B\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)\Delta t \tag{3.12}$$

$$\Delta B \approx \underbrace{C_w B_w - C_e B_e + C_s B_s - C_n B_n}_{\text{refer to 3.3.1}} + \underbrace{B\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)\Delta t}_{\text{refer to 3.3.2}} \tag{3.13}$$

Two clearly different methods are used to compute the contribution of advection to the change of $\vec{u}$ and are conveniently identified as "the first part" and "the second part" of the advection term. The former requires the use of the TVD Superbee method in order to be computed, while the latter can be found by using FDM.

3.3.1 Integration of the first part of the advection term

Note that the first part of the scheme is computed analogously to the way it was presented in section 3.2, as shown in expressions (3.14) and (3.15).

$$\begin{aligned}\Delta B_1 &= C_w B_w - C_e B_e + C_s B_s - C_n B_n \\ &= C_w^+ B_w^+ + C_w^- B_w^- - C_e^+ B_e^+ - C_e^- B_e^- + C_s^+ B_s^+ + C_s^- B_s^- - C_n^+ B_n^+ - C_n^- B_n^-\end{aligned} \tag{3.14}$$

$$\begin{aligned}
B_w^+ &= B_{i-1,j}^n + \tfrac{1}{2}\Psi\left(r_{i-1,j}^+\right)\left(1 - C_w^+\right)\left(B_{i,j}^n - B_{i-1,j}^n\right) && \text{with } r_{i-1,j}^+ = \tfrac{B_{i-1,j}^n - B_{i-2,j}^n}{B_{i,j}^n - B_{i-1,j}^n} \\
B_w^- &= B_{i,j}^n - \tfrac{1}{2}\Psi\left(r_{i-1,j}^-\right)\left(1 + C_w^-\right)\left(B_{i,j}^n - B_{i-1,j}^n\right) && \text{with } r_{i-1,j}^- = \tfrac{B_{i+1,j}^n - B_{i,j}^n}{B_{i,j}^n - B_{i-1,j}^n} \\
B_e^+ &= B_{i,j}^n + \tfrac{1}{2}\Psi\left(r_{i,j}^+\right)\left(1 - C_e^+\right)\left(B_{i+1,j}^n - B_{i,j}^n\right) && \text{with } r_{i,j}^+ = \tfrac{B_{i,j}^n - B_{i-1,j}^n}{B_{i+1,j}^n - B_{i,j}^n} \\
B_e^- &= B_{i+1,j}^n - \tfrac{1}{2}\Psi\left(r_{i,j}^-\right)\left(1 + C_e^-\right)\left(B_{i+1,j}^n - B_{i,j}^n\right) && \text{with } r_{i,j}^- = \tfrac{B_{i+2,j}^n - B_{i+1,j}^n}{B_{i+1,j}^n - B_{i,j}^n} \\
B_s^+ &= B_{i,j-1}^n + \tfrac{1}{2}\Psi\left(r_{i,j-1}^+\right)\left(1 - C_s^+\right)\left(B_{i,j}^n - B_{i,j-1}^n\right) && \text{with } r_{i,j-1}^+ = \tfrac{B_{i,j-1}^n - B_{i,j-2}^n}{B_{i,j}^n - B_{i,j-1}^n} \\
B_s^- &= B_{i,j}^n - \tfrac{1}{2}\Psi\left(r_{i,j-1}^-\right)\left(1 + C_s^-\right)\left(B_{i,j}^n - B_{i,j-1}^n\right) && \text{with } r_{i,j-1}^- = \tfrac{B_{i,j+1}^n - B_{i,j}^n}{B_{i,j}^n - B_{i,j-1}^n} \\
B_n^+ &= B_{i,j}^n + \tfrac{1}{2}\Psi\left(r_{i,j}^+\right)\left(1 - C_n^+\right)\left(B_{i,j+1}^n - B_{i,j}^n\right) && \text{with } r_{i,j}^+ = \tfrac{B_{i,j}^n - B_{i,j-1}^n}{B_{i,j+1}^n - B_{i,j}^n} \\
B_n^- &= B_{i,j+1}^n - \tfrac{1}{2}\Psi\left(r_{i,j}^-\right)\left(1 + C_n^-\right)\left(B_{i,j+1}^n - B_{i,j}^n\right) && \text{with } r_{i,j}^- = \tfrac{B_{i,j+2}^n - B_{i,j+1}^n}{B_{i,j+1}^n - B_{i,j}^n}
\end{aligned} \tag{3.15}$$

Note that because the variables are not centred in the CVs, the values of the Courant numbers are *not* computed as in (3.8). Instead, the expressions used to determine their value are presented in (3.16) and (3.17) and are derived from Figure 3.7 and Figure 3.8, respectively.

In Shallo Worlds, `solve_adv_u(...)` computes the required Courant numbers with the expressions below, which are found by creating a cell around the staggered u, as shown in Figure 3.7.

$$
\begin{aligned}
C_w^+ &= 0.5 \cdot \left(\frac{u_{i-1,j} + \left|u_{i-1,j}\right|}{2} + \frac{u_{i,j} + \left|u_{i,j}\right|}{2} \right) \cdot \frac{\Delta t}{\Delta x} \\
C_w^- &= 0.5 \cdot \left(\frac{u_{i-1,j} - \left|u_{i-1,j}\right|}{2} + \frac{u_{i,j} - \left|u_{i,j}\right|}{2} \right) \cdot \frac{\Delta t}{\Delta x} \\
C_e^+ &= 0.5 \cdot \left(\frac{u_{i,j} + \left|u_{i,j}\right|}{2} + \frac{u_{i+1,j} + \left|u_{i+1,j}\right|}{2} \right) \cdot \frac{\Delta t}{\Delta x} \\
C_e^- &= 0.5 \cdot \left(\frac{u_{i,j} - \left|u_{i,j}\right|}{2} + \frac{u_{i+1,j} - \left|u_{i+1,j}\right|}{2} \right) \cdot \frac{\Delta t}{\Delta x} \\
C_s^+ &= 0.5 \cdot \left(\frac{v_{i,j-1} + \left|v_{i,j-1}\right|}{2} + \frac{v_{i+1,j-1} + \left|v_{i+1,j-1}\right|}{2} \right) \cdot \frac{\Delta t}{\Delta y} \\
C_s^- &= 0.5 \cdot \left(\frac{v_{i,j-1} - \left|v_{i,j-1}\right|}{2} + \frac{v_{i+1,j-1} - \left|v_{i+1,j-1}\right|}{2} \right) \cdot \frac{\Delta t}{\Delta y} \\
C_n^+ &= 0.5 \cdot \left(\frac{v_{i,j} + \left|v_{i,j}\right|}{2} + \frac{v_{i+1,j} + \left|v_{i+1,j}\right|}{2} \right) \cdot \frac{\Delta t}{\Delta y} \\
C_n^- &= 0.5 \cdot \left(\frac{v_{i,j} - \left|v_{i,j}\right|}{2} + \frac{v_{i+1,j} - \left|v_{i+1,j}\right|}{2} \right) \cdot \frac{\Delta t}{\Delta y}
\end{aligned}
\tag{3.16}
$$

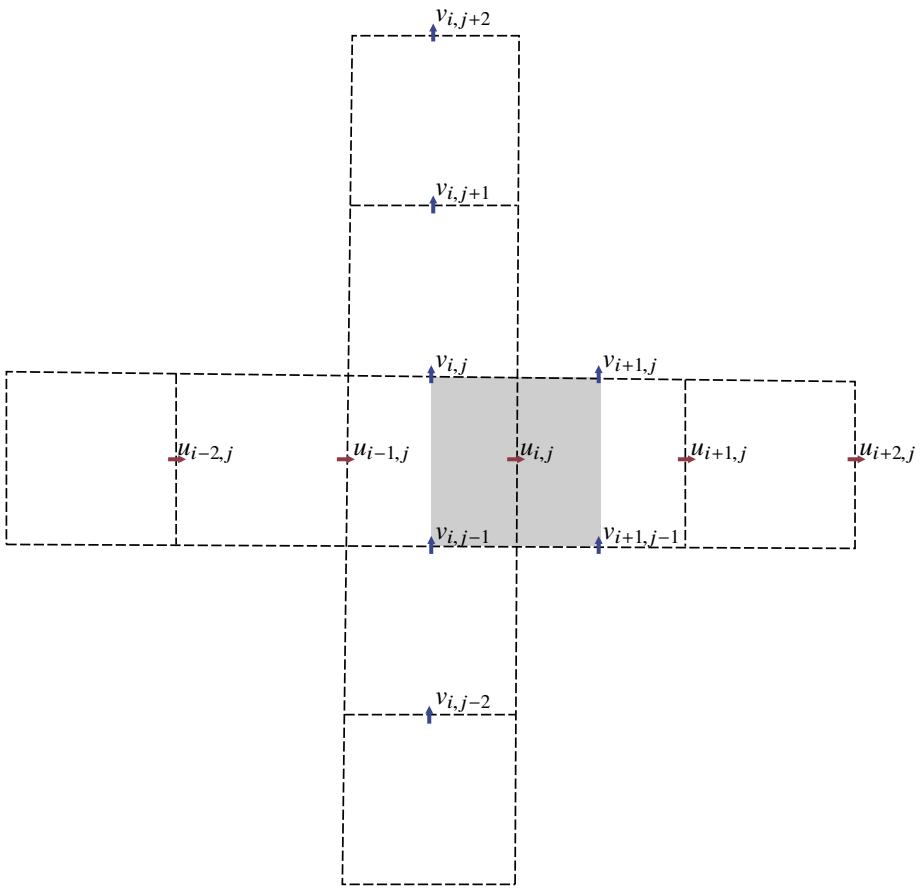


Figure 3.7: Grid cell staggered in the x direction

As for the contribution of the advection term to v, `solve_adv_v(...)` of Shallow Worlds uses the following expressions in its TVD Superbee scheme, which are derived from the situation in Figure 3.8

$$
\begin{aligned}
C_w^+ &= 0.5 \cdot \left(\frac{u_{i-1,j+1} + \left|u_{i-1,j+1}\right|}{2} + \frac{u_{i-1,j} + \left|u_{i-1,j}\right|}{2} \right) \cdot \frac{\Delta t}{\Delta x} \\
C_w^- &= 0.5 \cdot \left(\frac{u_{i-1,j+1} - \left|u_{i-1,j+1}\right|}{2} + \frac{u_{i-1,j} - \left|u_{i-1,j}\right|}{2} \right) \cdot \frac{\Delta t}{\Delta x} \\
C_e^+ &= 0.5 \cdot \left(\frac{u_{i,j+1} + \left|u_{i,j+1}\right|}{2} + \frac{u_{i,j} + \left|u_{i,j}\right|}{2} \right) \cdot \frac{\Delta t}{\Delta x} \\
C_e^- &= 0.5 \cdot \left(\frac{u_{i,j+1} - \left|u_{i,j+1}\right|}{2} + \frac{u_{i,j} - \left|u_{i,j}\right|}{2} \right) \cdot \frac{\Delta t}{\Delta x} \\
C_s^+ &= 0.5 \cdot \left(\frac{v_{i,j-1} + \left|v_{i,j-1}\right|}{2} + \frac{v_{i,j} + \left|v_{i,j}\right|}{2} \right) \cdot \frac{\Delta t}{\Delta y} \\
C_s^- &= 0.5 \cdot \left(\frac{v_{i,j-1} - \left|v_{i,j-1}\right|}{2} + \frac{v_{i,j} - \left|v_{i,j}\right|}{2} \right) \cdot \frac{\Delta t}{\Delta y} \\
C_n^+ &= 0.5 \cdot \left(\frac{v_{i,j} + \left|v_{i,j}\right|}{2} + \frac{v_{i,j+1} + \left|v_{i,j+1}\right|}{2} \right) \cdot \frac{\Delta t}{\Delta y} \\
C_n^- &= 0.5 \cdot \left(\frac{v_{i,j} - \left|v_{i,j}\right|}{2} + \frac{v_{i,j+1} - \left|v_{i,j+1}\right|}{2} \right) \cdot \frac{\Delta t}{\Delta y}
\end{aligned}
\tag{3.17}
$$

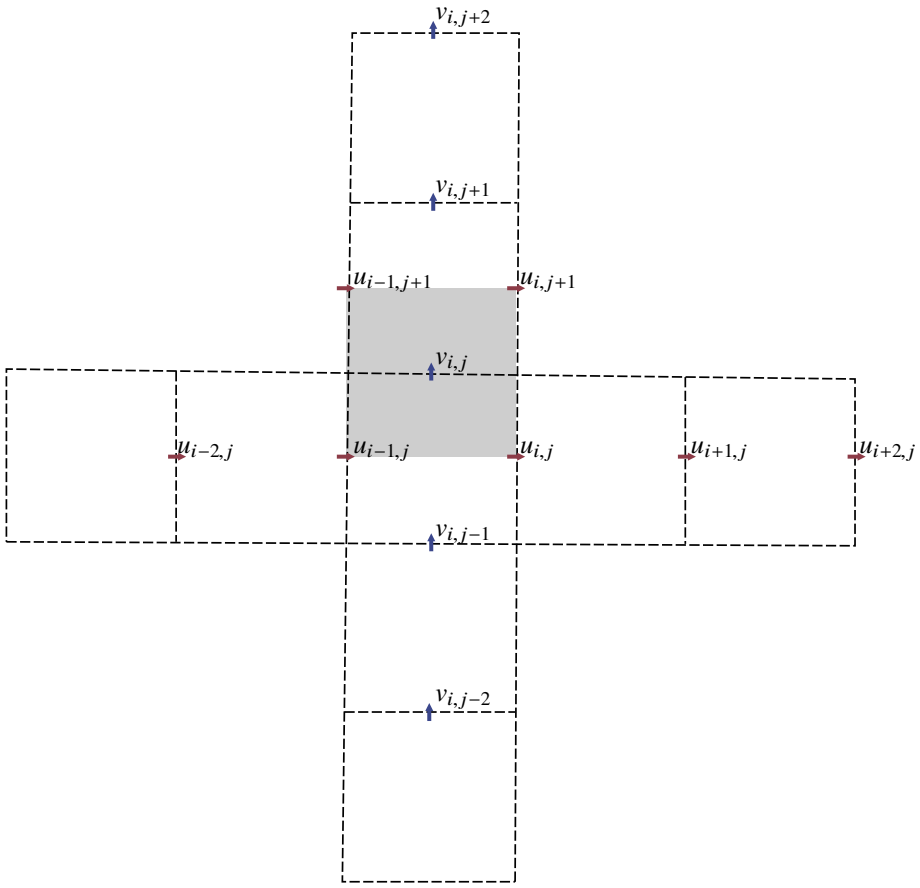


Figure 3.8: Grid cell staggered in the y direction

3.3.2 Integration of the second part of the advection term

The second part of the advection can be integrated according to a simple time-forward iteration[6]. Departing from (3.18), the expressions in (3.19) are found.

$$\Delta B_2 = B\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) \tag{3.18}$$

$$\begin{cases} \Delta u_2 = u_{i,j} \cdot \dfrac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} + u_{i,j} \cdot \dfrac{\frac{v_{i,j}+v_{i+1,j}}{2} - \frac{v_{i,j-1}+v_{i+1,j-1}}{2}}{2\Delta y} \\ \Delta v_2 = v_{i,j} \cdot \dfrac{\frac{u_{i,j}+u_{i,j+1}}{2} - \frac{u_{i-1,j}+u_{i-1,j+1}}{2}}{2\Delta x} + v_{i,j} \cdot \dfrac{v_{i,j+1} - u_{i,j-1}}{2\Delta y} \end{cases} \tag{3.19}$$

The equations above have been implemented in `solve_adv_u(...)` and `solve_adv_v(...)`.

3.4 Integration of the pressure term

The influence of gradient of pressure on the velocity

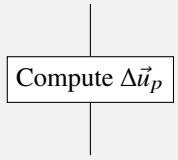


Figure 3.9: $\vec{u}_\mathrm{p}$ integration block

A system where the only cause of temporal change of $\vec{u}$ is the pressure distribution, is characterized as

$$\begin{cases} \dfrac{\partial u}{\partial t} = -g\dfrac{\partial \eta}{\partial x} \\ \dfrac{\partial v}{\partial t} = -g\dfrac{\partial \eta}{\partial y} \end{cases} \tag{3.20}$$

The contribution of the pressure distribution to the change of velocity can be computed in Shallow Worlds by calling `solve_pres_u(...)` and `solve_pres_v(...)` as part of the block in Figure 3.9 of the flowchart in Figure 3.1 .

The equations in (3.20) can be integrated using FDM as shown in the system of equations (3.20).

$$\begin{cases} \Delta u_p = -g\dfrac{\eta_{i+1,j} - \eta_{i,j}}{\Delta x}dt \\ \Delta v_p = -g\dfrac{\eta_{i,j+1} - \eta_{i,j}}{\Delta y}dt \end{cases} \tag{3.21}$$

This expressions have been implemented in `solve_pres_u(...)` and `solve_pres_v(...)`.

3.5 Adjusting the integrated values

In order to achieve precise results throughout time, the Euler method, which assumes $\alpha^{n+1} = \alpha^n + \Delta\alpha$, should not be used. Instead, a multistep method such as the 2nd order or 3rd order Adams–Bashforth shall be used in blocks like the one presented in Figure 3.10 of figure Figure 3.1.

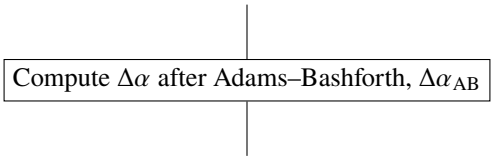


Figure 3.10: The Adams–Bashforth block of the solver core, as presented in Figure 3.1

The 3rd order Adams–Bashforth method for constant Δt has been implemented in Shallow Worlds as `do_AdamsBashforth(...)` and is shown in (3.22).

$$\begin{cases} \Delta\alpha_{\mathrm{AB}} = \Delta\alpha^n & \text{for the first time step} \\ \Delta\alpha_{\mathrm{AB}} = \frac{3}{2}\Delta\alpha^n - \frac{1}{2}\Delta\alpha^{n-1} & \text{for the second time step} \\ \Delta\alpha_{\mathrm{AB}} = \frac{23}{12}\Delta\alpha^n - \frac{4}{3}\Delta\alpha^{n-1} + \frac{5}{12}\Delta\alpha^{n-2} & \text{for the third time step} \end{cases} \tag{3.22}$$

The 3rd order Adams–Bashforth method is more precise than the 2nd order Adams–Bashforth or the Euler method, as it can be seen in figure Figure 3.11 for a initial value problem defined by $y' = y$ and $y(0) = 1$.

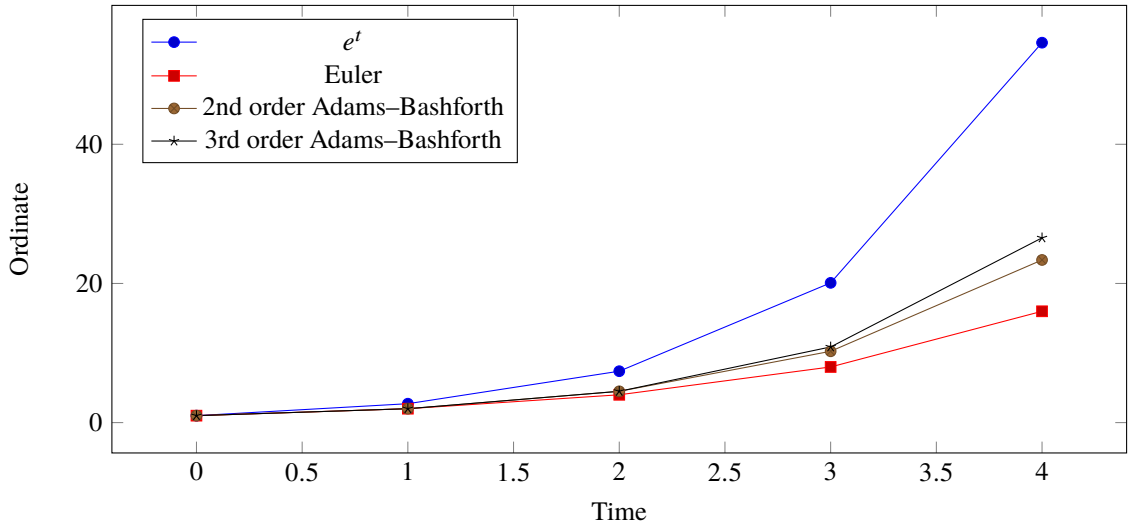


Figure 3.11: Comparison of the Euler and Adams–Bashforth methods.

The reader may want to check out `do_AdamsBashforth_variable_Dt(...)` for a expression of the 2nd order Adams–Bashforth method with variable Δt. Arnau Sabatés, with the help of the author, found the 3rd order Adams–Bashforth expression for variable increments of time. Because the current version of Shallow Worlds only supports constant increments of time and the differences of using a 3rd order Adams–Bashforth over the 2nd order Adams–Bashforth are negligible, the demonstration was lost.

If no geometrical factors or planet rotation were present, $\alpha^{n+1} = \alpha^n + \Delta\alpha_{\mathrm{AB}}$. In Shallow Worlds, however, this variable has been called $\alpha^* = \alpha^n + \Delta\alpha_{\mathrm{AB}}$, because one more step (see section 3.6) is required before finding α^{n+1}.

3.6 Taking into account the Coriolis factor and geometrical parameters

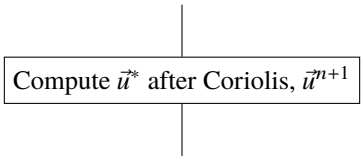


Figure 3.12: $\vec{u}$ with Coriolis force integration block of the solver core, as presented in Figure 3.1

The contribution of the planet's angular speed to $\vec{u}$ has not been taken into account to this point. The functions `solve_Coriolis_u(...)` and `solve_Coriolis_v(...)` of Shallow Worlds implement a method with which $\vec{u}^{n+1}$ is finally computed from the current $\Delta\vec{u}$ and $\vec{u}^*$. This functions include the geometrical factors that arise from the use of spheroidal coordinates.

3.7 The role of the boundary conditions

The BCs are imposed before the start of the loop and at the end of the integration of the equations. The BCs implemented in the last version of Shallow Worlds are the periodic BC and the channel BC, as these are the currently needed for the study of atmospheric phenomena.

3.7.1 The periodic boundary condition

The periodic BC specifies that the incoming flow must be the same as the outgoing flow. This means that the halo must be filled with the values of the opposite side of the domain, i.e. for the horizontal velocity u,

the periodic BC is imposed as shown in (3.23).

$$\begin{aligned} u_{-1,j} &= u_{n_x,j} \\ u_{0,j} &= u_{n_x-1,j} \\ u_{n_x+1,j} &= u_{1,j} \\ u_{n_x+2,j} &= u_{2,j} \end{aligned} \tag{3.23}$$

Note that $u_{-1,j}$, $u_{0,j}$, u_{n_x+1} and u_{n_x+2} are part of the halo and *not of the domain*. For the sake of the explanation, the indices of the elements of the domain range from 1 to n_x for the x direction and from 1 to n_y for the y direction. The indices for the whole matrix—which includes the halo—range from -1 to $n_x + 2$ and from -1 to $n_y + 2$ for the x and y direction, respectively. The complement of the domain in the matrix is the halo, which has been drawn as a hollow rectangle in Figure 3.13.

For example, the horizontal velocity u field is stored in matrix form as shown in figure Figure 3.13. Variables such as coordinates are stored in the same fashion.

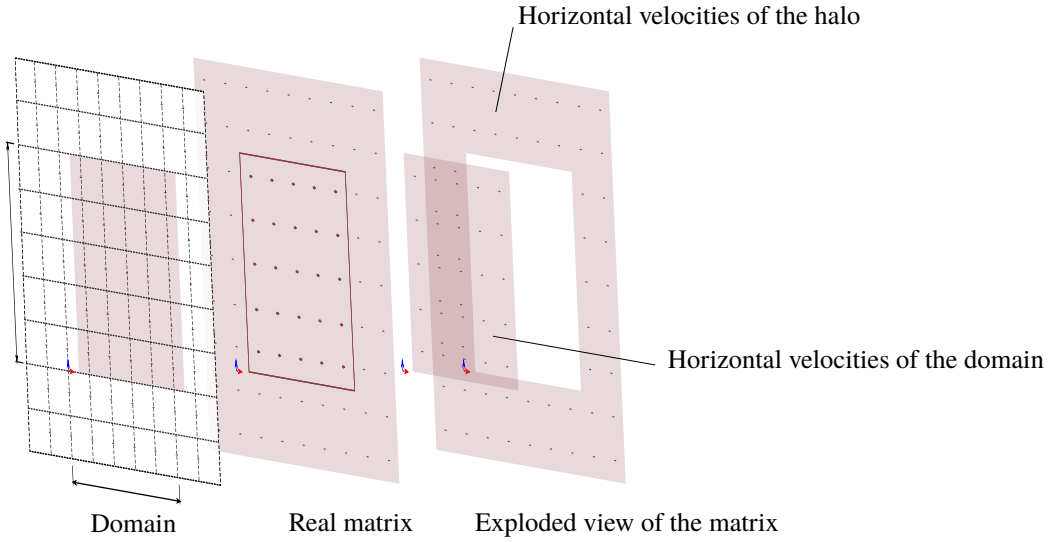


Figure 3.13: Arrangement of u in a matrix

`halo_update(...)` is in charge of fulfilling the periodic boundary condition, as explained in subsection 5.4.1.

3.7.2 The channel boundary condition

The staggered grid plays an important role in the definition of the channel BC and in fact, one must treat differently the top and bottom boundaries. This boundary condition states that there is no horizontal velocity u relative to the boundary (no-slip condition) and no vertical velocity v (wall condition). Technically, this means that the tangential velocity at the wall is that of the wall and that the normal velocity is 0. In order to achieve this, `do_channel(...)` contains the following in (3.24) and (3.25) in its core. The tangential velocities u include the zonal wind, if present.

Top boundary

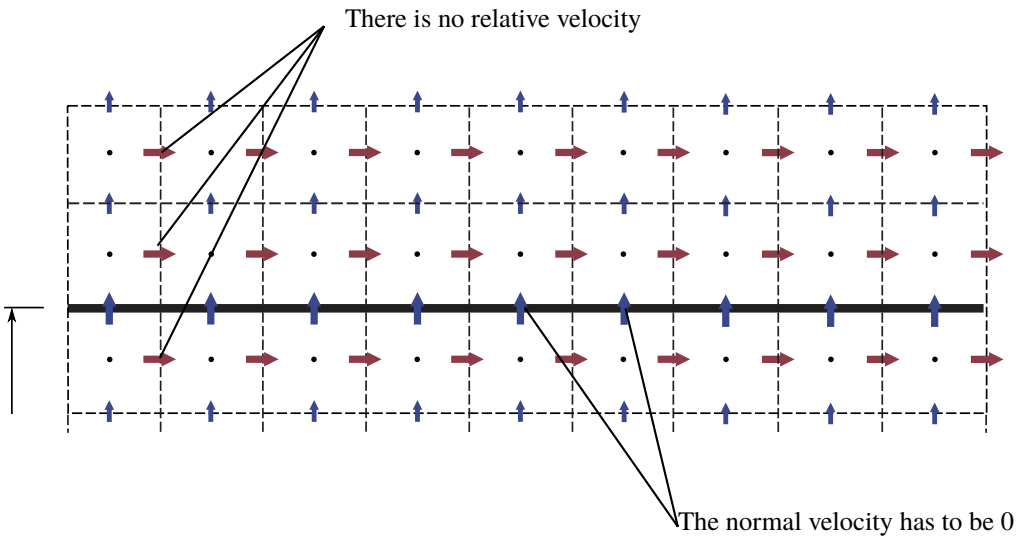


Figure 3.14: Detail of the top boundary

The expressions in (3.24) have been derived from the discretization of u and v around the top boundary (see Figure 3.14).

$$\begin{cases} u_{i,n_y+1} = u_{i,n_y} \\ u_{i,n_y+2} = u_{i,n_y+1} \\ v_{i,n_y} = 0 \\ v_{i,n_y+1} = 0 \\ v_{i,n_y+2} = 0 \end{cases} \tag{3.24}$$

Bottom boundary

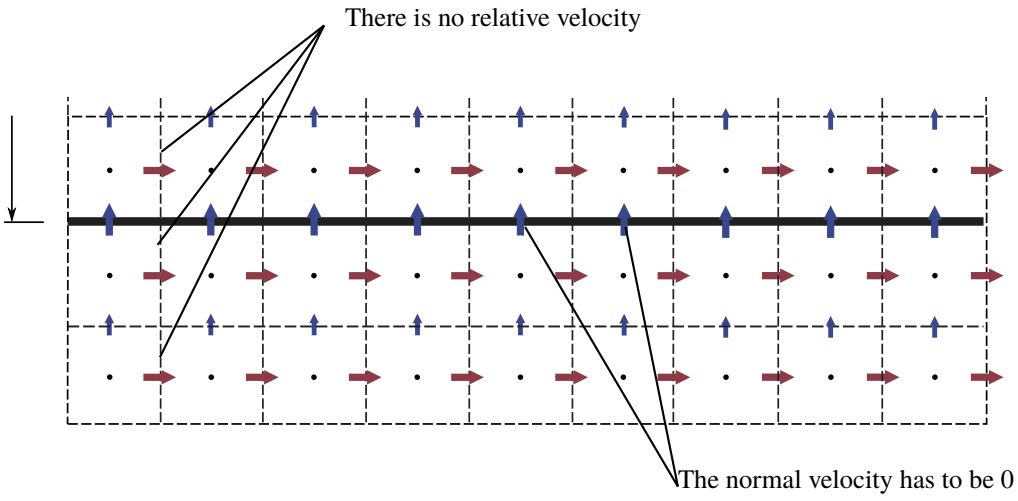


Figure 3.15: Detail of the bottom boundary

The expressions in (3.25) have been derived from the discretization of u and v around the bottom boundary (see Figure 3.15).

$$\begin{cases} u_{i,0} = u_{i,1} \\ u_{i,-1} = u_{i,0} \\ v_{i,0} = 0 \\ v_{i,-1} = 0 \end{cases} \tag{3.25}$$

CHAPTER 4

Validation of the solver

In this chapter, the validation process of Shallow Worlds is described. Validating the code is an extremely important task and consequently, different methods have been used at the different stages of the solver. This job could not have been realized without the help of Arnau Sabatés, who supplied results that were compared with the output of Shallow Worlds. In his thesis[8], the reader will find extended descriptions of the methods presented in this chapter.

The aim of this chapter is to show how the output of a solver can be analysed and validated. It should be understood as a reference for future validations of solvers and not only as an explanation of the methodology followed during this FYP.

4.1 Validation of the Matlab prototype

Before the start of the development of Shallow Worlds (see chapter 6), a prototype solver was created in Matlab. This prototype was used as a learning platform for the shallow water equations as well as for the validation methods, which will be the subject of this method.

The Prototype only included the features that allowed it to solve cases equivalent to Shallow Worlds's `sw.probdef=DROP`. These features were validated using two methods, which are explained in the following sub-sections.

4.1.1 Validation of the Prototype with The Method of Manufactured Solutions

The Method of Manufactured Solutions (MMS) consists on performing a simulation with a particular set of initial parameters that are solution for the shallow water equations. The program is tested successful if the differences between the analytic solution and the solver's output are negligible. Otherwise, the program has an error. The method is set up according to the information delivered in Arnau Sabatés's thesis[8].

The Shallow Worlds MMS validation was carried out step by step because, as seen in Figure 3.1, the solver is built on top of blocks that are sequentially executed. By validating each of the blocks in the order they appear in the flowchart, it can be confidently said and demonstrated that the software outputs the correct result.

This method is extremely powerful but it is also complex to implement in some solvers because source terms have to be added to the equations and other terms of the equations have to be modified accordingly. This might lead to changes of the integration routines and of the solver's main loop, but it was not the case of the Prototype. MMS can be easily implemented in a Matlab code thanks to Matlab's Symbolic Math Toolbox but becomes difficult to use with C, as it does not have symbolic expression capabilities by default.

Because of the start of Shallow Worlds, the author validated all the blocks implemented in the Prototype with the exception of the Coriolis step. The results after the Coriolis were later verified against Arnau Sabatés's, who had validated his with MMS.

4.2 Validation of Shallow Worlds

Shallow Worlds started its development while Arnau Sabatés performed validation tasks. By the end of the development of the solver in C, correct results were already available thanks to good coordination.

4.2.1 Comparison of the output of two solvers

If two programs that solve the same set of equations are available, the accuracy of the output can be evaluated by comparing the solution of the same initial value problem. While this method cannot be used to conclude if the result is correct or not, it may help in the debugging task. Moreover, if the development of both programs has been followed by both authors, the method may be used to decide if the solver has been implemented as expected.

The output of both the Prototype and Shallow Worlds has been compared to that of Arnau Sabatés solver and, at the time of writing, *the solvers output practically the same results under the same initial conditions*, on orthogonal and spheroidal coordinate systems. This is the method that has been used the most in the validation tasks of Shallow Worlds from the very beginning, as MMS could not be implemented without time delays.

The current discrepancies between the output of Arnau Sabatés's solver and Shallow Worlds are due to the order in which the operations are carried out in the core of Matlab and C code. The use of MPI may also contribute to increasing this differences.

Both Shallow Worlds and Arnau Sabatés's solvers have to be compared to the legacy Shallow Worlds.

4.2.2 Visual inspection of the results

Even though that through visual inspection the accuracy of the results cannot be rated with rigour, if a simulation clearly outputs an incoherent solution from a known set of initial conditions, an error has probably happened.

By using this method on some test simulations of Arnau Sabatés, an error of indices was found. In his plot, ripples at the boundaries of the colour bands (of a potential vorticity Π_s plot) appeared as a storm moved leftwards and were intensified when the storm reached the boundary. This lead him to think that a problem with the indices in his implementation of the periodic BC were not correct. Once this was amended, his prototype in Matlab became the solver against which Shallow Worlds had to be verified (if the legacy Shallow Worlds is not considered).

This method cannot only be used to validate the solver, but also the post-processing tools. An error of the SWreader was also detected after plotting the solver results in ParaView, as it has been explained in see section 7.2.

CHAPTER 5

A presentation of the solver's structure

Shallow Worlds has been developed to support multiprocessor computing in order to simulate large domains efficiently, as the goal is to a deliver *scalable* high performance computing (CVHPC), see=[Glossary:]HPCg code that will even work on a supercomputer such as Barcelona Supercomputing Center (BSC)'s MareNostrum.

While low-level *parallel programming* is complex, a good program design makes it almost as easy as to program sequential software. The solver has been built on top of `sppde`, a collection of macros and functions programmed by Manel Soria[9] with the aim to facilitate MPI–parallel programming.

Even though Shallow Worlds is structured in a fashion that makes it as easy to add features as to modify the existing ones, it heavily relies on the `sppde` library so the solver core cannot be understood without this library.

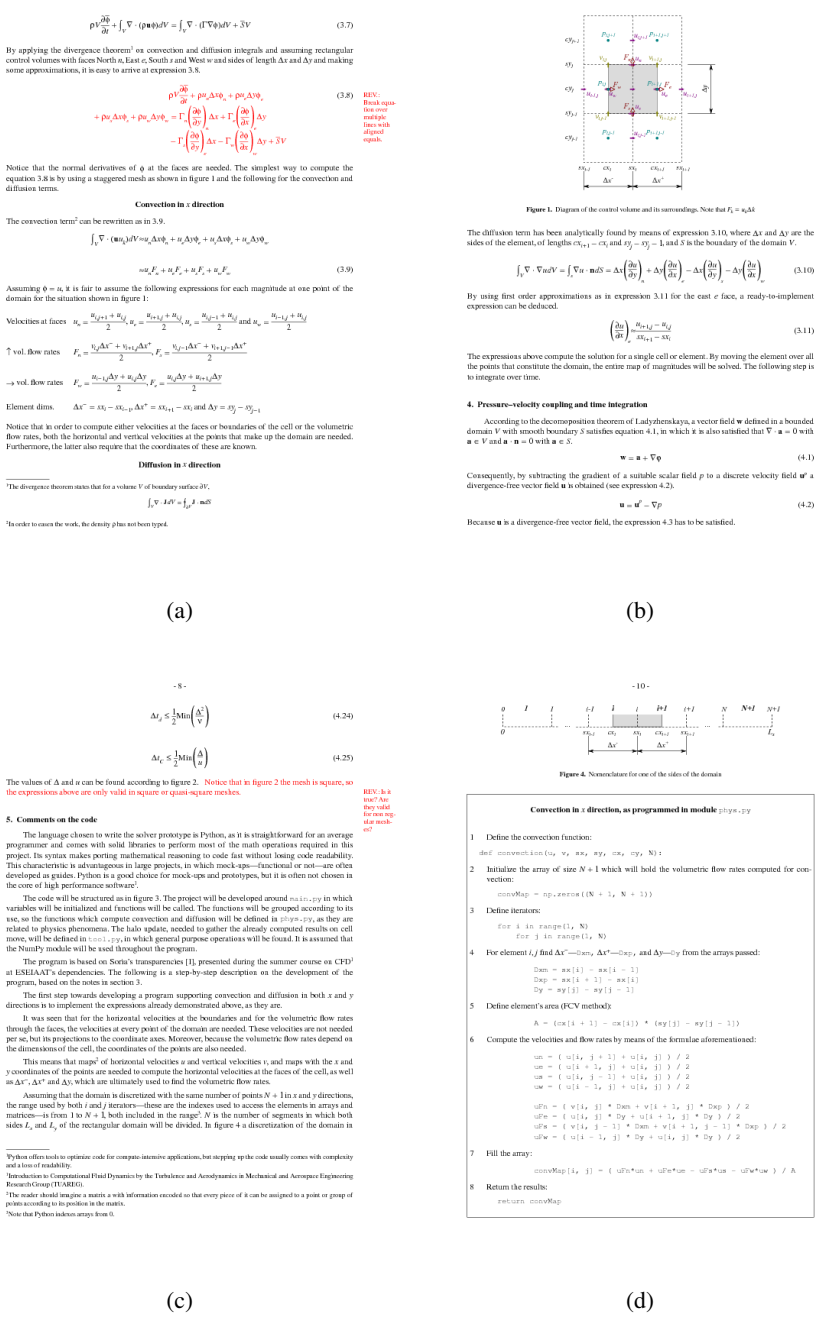
In the direction of achieving fast computing, C has been chosen as the core programming language. This language is still broadly used at the core of high performance software together with Fortran. Despite the rising of newer programming languages with extremely attractive features such as Julia, both Fortran and C are already consolidated in terms of documentation, syntax and available libraries. MPI use in a language other than Fortran or C is usually achieved through packages that provide bindings to the Fortran or C libraries, which may be error-prone.

5.1 The author's previous experience with a similar solver

During the "Introduction to CFD" summer course[10], the author was assigned to develop a simple NSE solver. This solver was to be in many ways similar to Shallow Worlds. It was written in Python instead of C or Matlab.

Because of time and difficulty requirements the solver was never fully developed but the exercise led to a good understanding of the inner workings of a basic solver.

Documentation related to the project was written in parallel to the programming of the solver. The reader can find some of the pages in Figure 5.1, while the files created during the project development in the author's GitHub repository. Note that the documentation is written in Lout.



(a)

(b)

(c)

(d)

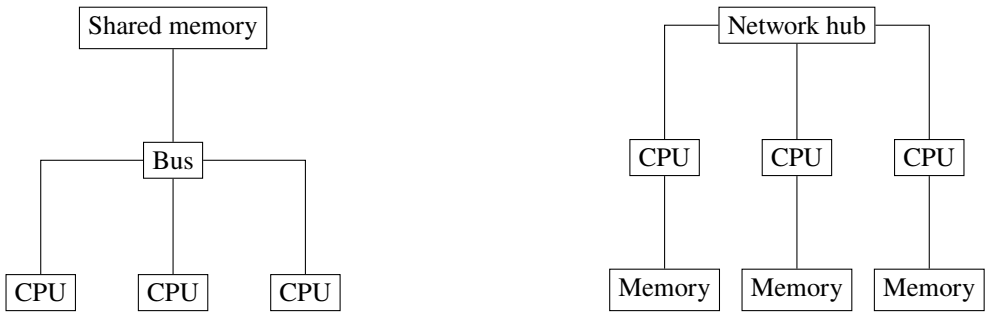
Figure 5.1: Pages of the documentation of the solver developed for the summer course

5.2 An introduction to parallel computing

Parallel computing refers to the use of multiple processors to simultaneously carry out subtasks of a main task with the goal to finish the job earlier than a single processor. Generally, *parallelism* refers to the fact that the subtasks are executed at the same time by the participants. Usually, they have to exchange values through the network to carry on with the subtask and consequently, a job that is executed in parallel is often not executed individually by the processors.

HPC infrastructures have different architectures when it comes to memory management. In general, multiprocessor systems can either share a single memory space or have private memory spaces for each

of the processors[1]. Depending on the network, these are called *shared memory* machines or *distributed memory* machines, respectively (see Figure 5.2).



(a) A shared memory system

(b) A distributed memory system

Figure 5.2: Multiprocessor systems

The variables used in programs designed to work in architectures such as the one presented in Figure 5.2a are found by all the processors in the same address in memory, as opposed to distributed memory systems, where every one runs the same code independently, storing the variables in their own private memory. When data needs to be passed to other processors, a message passing infrastructures are needed.

Shallow Worlds uses MPI for data exchange as its goal is to establish a portable, efficient and flexible standard for message passing, for use in HPC applications. There are different implementations of the standard, one of them being the Open MPI library, which has been undergoing development since 2004. As presented in the standard, the library defines functions such as

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm)
```

and

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm
    comm, MPI_Status *status)
```

These functions are defined in the standard, but the inner workings of these depend on the implementation and therefore one may encounter different behaviours with different libraries and/or operating systems.

The definitions above are used to send and receive data to and from other processors, each one running its own "copy" of the program, and thereby producing its own results. When a processor *sends* a variable, another processor has to be in charge of listening and receiving the function. Poorly designed parallel software may lead to *deadlocks* and other undesired errors, which may be very difficult to debug and resolve.

5.3 The Structured Parallel PDE solver—sppde library: parallel programming made easy

The Structured Parallel PDE solver–`sppde` library contains a set of functions and macros that can be used to create a solver for situations where a structured grid is used. The `sppde` is built on top of the structure shown in Listing 5.1, which contains information of a given processor's region of the domain. Further information on the division of the domain is found in subsection 5.4.1.

Listing 5.1: Basic structure of `sppde`

```
typedef struct {
        int sh; // Halo size (>=1)
        int nd; // Number of dimensions 2 or 3
        int l0[4]; // Start index of the subdomain in every direction (the first number
            is not used)
        int l1[4]; // End index of the subdomain
        int gl0[4]; // Start index of the domain
```

[1]For the sake of the explanation, distributed shared memory computer systems will be not introduced.

```
        int gl1[4]; // End index of the domain
        int al0[MAXP][4];
        int al1[MAXP][4];
        int s[4];
        int S;
        int gpc[4]; // Number of procs per dimension
        int pc[4]; // Processor coordinates in the global grid (starting from (1,1)) (
            row-major)
        int nb_e,nb_w,nb_n,nb_s; // Neighbouring processors
        int nb_t,nb_b;
        int bc0[4];
        int bc1[4];
        int per[4]; // Periodicity in each direction
} map;
```

It can be seen that the processor has knowledge of its position in the processor grid (see Figure 5.3), the start and end indices of its subdomain and the whole domain, as well as the domain's periodicity. From this variables, a variety of macros are defined, of which the most commonly used are:

`check(...)` Indispensable for debugging.

`forall(...)` Covers every point of the subdomain without specifying local limits.

`forallh(...)` Similar to `forall(...)`, but the halo is also covered.

`ac(...)` Accesses a position i, j of a matrix with the intrinsic properties that are defined in `map`.

`ifowned(...)` Checks whether a given point of the domain falls in the processor's subdomain.

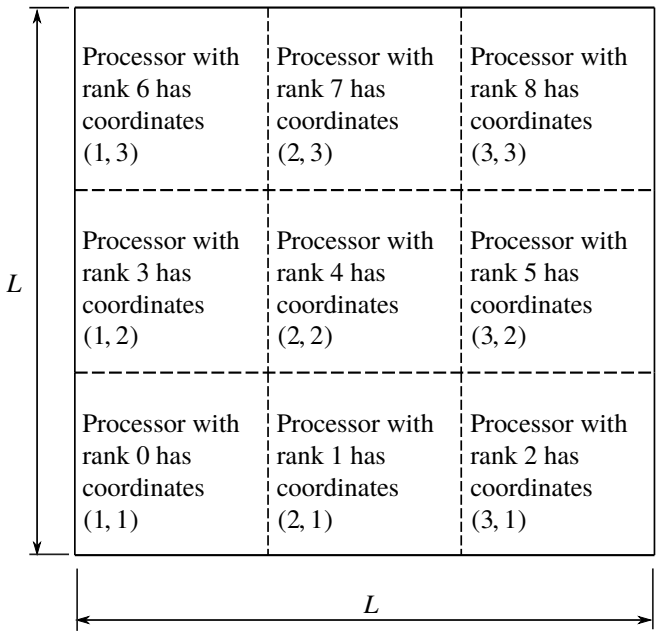


Figure 5.3: Processor world. Note that the coordinates shown are the values of the second and third element of `map.pc[4]` and in any case are they the domain i, j points.

`sppde` comes in two files, `sppde-core.c` and `sppde-hupdate.c`. In the former, map modifying, printing and debugging functions are defined. In the latter, all the halo-related and map gathering tools. A new `sppde-extensions.c` was created by the author of this project and it contains functions that compute the cumulative sum of a variable as well as the new `halo_update(...)`. These functions have not been thoroughly used but have passed the tests confronted with.

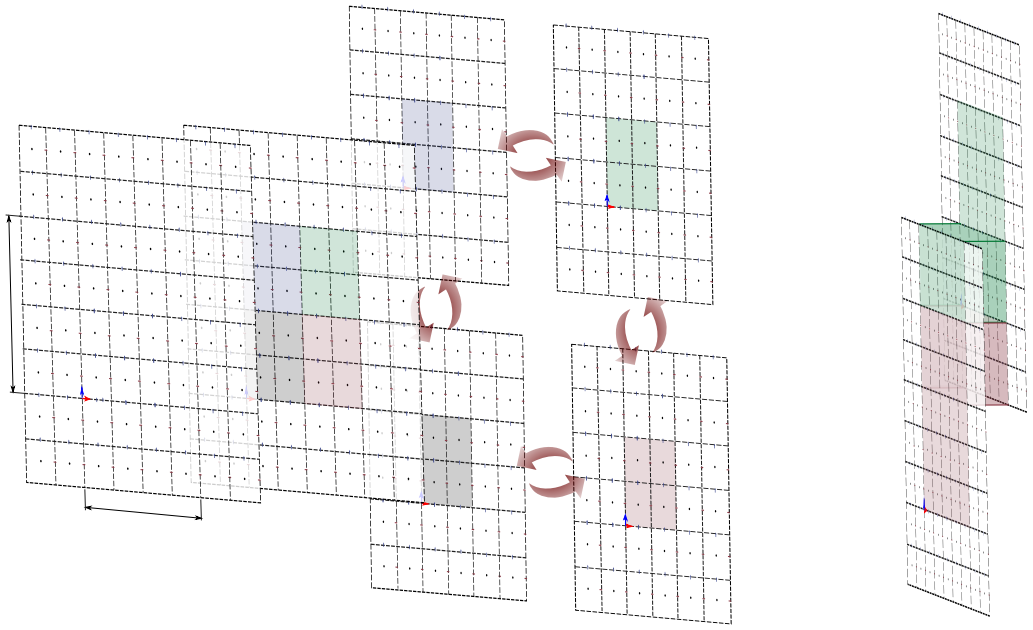
Thanks to these macros and the related functions, the Shallow Worlds source code can be read with almost the same ease as a typical sequential program. The essentials of the solver are further explained in the next section.

5.4 Shallow Worlds parallel design

5.4.1 The processor's halo, domain and subdomain

The distribution of the points of the domain among processors is done according to an input. The user decides the number of processors per side of the domain and the points are divided as uniformly as possible between processors. Every point is assigned a colour (not in the strict sense) to identify the subdomains. Any given processor only knows about its local limits (the memory is distributed) and cannot access any variable of a point it does not have. The points that are distributed are those of the domain, which does not include the halo, as shown in Figure 3.13 and Figure 5.4a.

Once the subdomains are specified, the processors then have to carry out the operations in every region independently. Because of the FDM and TVD requirements, a halo also needs to be created around their subdomains. This newly created halos may or may not overlap with the domain's halo. For example, the subdomain of a processor that falls in the middle of the domain will be surrounded by a halo that will not share any variable with the domain's halo.



(a) From the domain to the subdomains

(b) Halo overlap

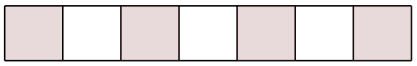
Figure 5.4: Distribution of the domain among processors

While the data of the halo of the original domain is updated according to BC (see section 2.4), the data of the halo of the subdomains is passed between adjacent processors so they solve the subdomain with the most updated result of the neighbours. In other words, the BC of the subdomains are not updated with the BC of the whole domain but with the results of the past iteration of the neighbours (for not bounding processors). The arrows in Figure 5.4a represent the exchange of information between neighbouring processors, whose halos overlap, as shown in Figure 5.4b.

The function that carries out this exchange of information is `halo_update(...)` (part of `sppde`) which calls `halo_update_y(...)` and `halo_update_x(...)`. These functions were originally written in terms of `MPI_Send(...)` and `MPI_Recv(...)`, which meant that information was exchanged in a direction at a time, e.g. the processor first sends its data to the processor to the north and then receives data from the processor to the south. The old `halo_update(...)` was rewritten in terms of `MPI_Sendrecv(...)`, which takes advantage of full-duplex networks which has increased the speed of the program. The old functions are still present in the code as `legacy_halo_update(...)` while the new functions can be found in `sppde_extensions.h` and `sppde_extensions.c`.

The treatment of the halos is different when the domain is periodic in a given direction. A periodic domain must satisfy that the values of one side of the domain are equal to those of the opposite site, as explained in section 2.4. The `halo_update(...)` function handles this situation by setting the boundary

processors as neighbours. Because the neighbour information that `halo_update(...)` uses is created from the parity of the processors in a given line, row or column, of the grid of processors (see Figure 5.3), the number of processors in a given axis with periodic BC has to be a multiple of two. In other words, in a line of a grid of processors, the even processors are assigned a role and the odd processors another. If the bounding processors act the same way, miscommunication leading to a deadlock may happen. A graphic description is shown in Figure 5.5.



(a) An odd number of processors may lead to a deadlock

(b) An even number of processors is needed in rows or columns with periodicity

Figure 5.5: Parity of the processors in a row. In a periodic domain, if the number of processors is odd in the direction of the periodicity, a deadlock may happen.

`halo_update(...)` also accounts for the case when the number of processors entered is 1. In this case, Shallow Worlds can also satisfy the periodicity boundary condition. It cannot, however, fulfil the periodicity condition if the global number of processors is different from 1 but in the direction the periodicity happens is. Even though that implementing this is immediate, the fact that `halo_update_single_map_y(...)` and `halo_update_single_map_x(...)` have not been proved to be efficient has delayed this improvement until further notice.

To sum up, an arrangement of 9 processors in the domain can be done as shown in Figure 5.3. Every region of the domain is characterized by a `map`, the structure presented in section 5.3. In the case of subdomains, properties such as processor coordinates are specified in `map m` and bBecause the memory is distributed, the values of these members might be different. The periodic BC cannot be used as in either directions the number of processors is odd.

5.4.2 Element indexing and memory management

Shallow Worlds provides a number of definitions that are set up to mimic matrices: η, u and v can be accessed in matrix notation with `ETA(...)`, `U(...)` and `V(...)`, which use `sppde`'s `ac(...)`. *Shallow Worlds indexes the elements of the matrix starting from number 1*, just like Julia or Matlab, so nodes and matrix positions start at $i, j = 1$ and end at $i = n_x, j = n_y$. Therefore, one might think that e.g. `ETA(nx/2,ny/2)` would return the value of η at the centre of the domain. However, it should be noted that because of MPI and the architecture in Figure 5.2b, the processors cannot access all of the variables of the domain. This means that only one processor has the value of η at $(n_x/2, n_y/2)$ and calling `ETA(nx/2,ny/2)` from a processor that does not have this point would cause the program to crash. Only the processor that has this point would return the correct value and continue the operation until it interacted with another processor (which had crashed before), causing errors such as deadlock. While there is a `get_val(...)` function to receive values from other processors, its use is not recommended because the body of the function makes use of computationally expensive operations. For reference, if the n_x-by-n_y domain was distributed among a grid of 2-by-2 processors, only the processor with rank 0 would have the desired value.

Multidimensional arrays or matrices are not stored as they are, but arranged in row-major order in the linear memory space. When the user accesses a point i, j of a n_x-by-n_y matrix through `ac(...)`, they is accessing the position $j + i \cdot (n_x - 1)$ of the dedicated memory space. This expression takes advantage of the fact that memory addresses initialized with `malloc(...)` are correlative, as seen in Figure 5.6.

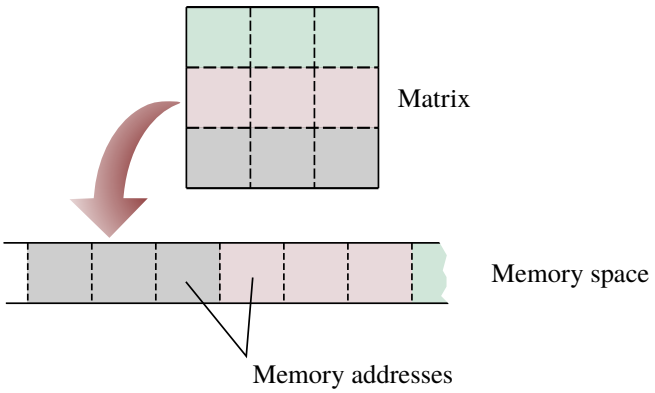


Figure 5.6: Memory arrangement of a matrix

Because of the design of the C programming language, arrays cannot have a variable number of elements at the time of compilation. Consequently, in order to have variable-size arrays and matrices, these have to be initialized with `malloc(...)` and terminated with `free(...)` when the variables are no longer required. A matrix of size n_x-by-n_y containing `double`s needs a memory space of $n_x \cdot n_y$ `double`s. The only arrays that are fixed-size are those that contain information of each of the three axes, such as `gl0[4]={0,1,1,1}`. The first number of this array, `gl0[0]`, is not used so `gl0[1]` contains information of the x axis and `gl0[2]` and `gl0[3]` of the y and z axes respectively. This criterion is used in all fixed-size matrices.

5.4.3 The sw structure

The solver follows the structure presented in Figure 3.1. In `sw_main.c` the reader can find a call to `create_world(...)` (which initializes the domain and subdomain variables), to `do_core_loop(...)` (which computes the desired result), and to `destroy_world(...)`.

Similarly to `sppde`, Shallow Worlds is build on top of a structure, which is presented in Listing 5.2 and in `sw`. `sw` is short for *shallow world* and is meant to store the simulation parameters, from the physical variables to subdomain information.

Listing 5.2: Basic structure of Shallow Worlds

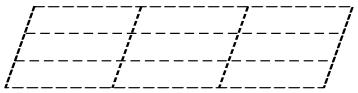
```
typedef struct {
        int probdef; // Problem definition, see above
        int nx, ny; // Number of cells in the centered mesh (there are nx+1 and ny+1
            division lines)
        double lon0,lon1; // Initial and final longitude
        double glat0,glat1; // Initial and final latitude
        double x0,x1; // Initial and final coordinates
        double y0,y1; // Initial and final coordinates
        map m; // Intrinsic properties of the domain
        map *M; // Pointer to m, to ease notation

        // Scalar values
        double alpha; // Value for hibrid schemes
        double Omega; // Angular velocity
        double rE,rP; // Equatorial and polar radii of the spheroid
        double epsilon2; // Radius ratio of the spheroid
        double mass; // Spheroid average mass
        double dens; // Density of the atmosphere
        double D; // Reference depth of the fluid layer

        // Scalar fields
        double *u,*v,*eta,*hB;
        double *zw;
        // double *lons,*lonc,*lonv,*glats,*glatc,*glatv;
        double *xs,*xc,*xv,*ys,*yc,*yv;
        double *Dxs,*Dxc,*Dxv,*Dys,*Dyc,*Dyv; // Distances between centered points and
            staggered points (centered, staggered, edge)
```

```
26          // double *xc,*xs,*yc,*ys; // Coordinates of the centered points and staggered
                points
27  } sw;
```

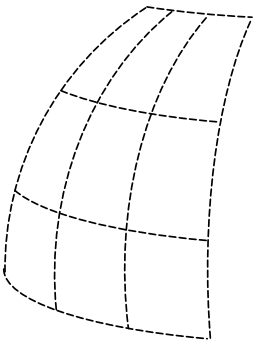
Note that `sw` contains coordinate information through `lon0`, `lon1`, `xc`, `xs`..., as well as information of the physical variables such as D, u, v, η or h_B, and the subdomain properties through `m`.

Initialization of the variables

The initialization of the variables is done by calling `create_world(...)`, defined in `sw_init.c`. The first variable to be initialized is `probdef`, which makes the solver to behave in a way or another. If `sw.probdef` is `DROP`, the solver will consider that e.g. the domain is periodic in both directions and that the system of coordinates to be used is orthogonal. If `sw.probdef` is `PLANET`, the solver will assume that the domain is a periodic channel in the x direction, that the coordinate system has to be initialized with `init_coords_spheroid(...)` and that there are notable increments in longitude and latitude.



(a) Representation of an orthogonal mesh



(b) Representation of an ellipsoidal mesh

Figure 5.7: Coordinate systems currently supported by Shallow Worlds

The shallow world `sw` has the needed coordinate information of the centred, staggered and vertex points, both in angular and spatial (on the domain's surface) distances. The different functions that are called at the core of the solver use the latter at their definitions. The convention that has been assumed is shown in Figure 5.8 and rose from the need to support the coordinate systems in Figure 5.7. While the sides of the CV may be of constant length, this is not the case for spheroidal coordinates and therefore, the sides and points need to be individually specified.

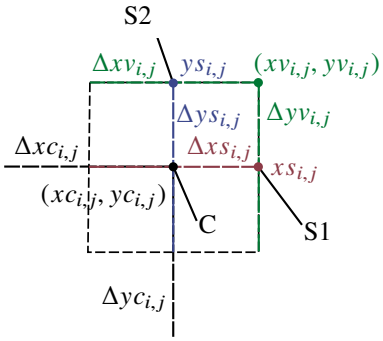


Figure 5.8: Coordinate-related variables of a given cell

All of the variables presented in Figure 5.8 are stored in matrices characterized by `map m`. In `create_world(...)`, the matrix for e.g. Δxv is initialized by calling `dmem(...)` like `sw.Dxv=dmem(sw.M)`. $\Delta xs_{i,j}$ is conveniently accessed in Shallow Worlds by means of `DXS(i,j)`, and $xc_{i,j}$ through `XC(i,j)`. The reader should know that, while spatial coordinates are stored in memory[2], longitudes and latitudes are

[2]Computing the coordinates on the surface of an ellipsoid is a very expensive operation (see `init_coords_spheroid(...)`) so creating the coordinates on-demand is not a good practice.

not. This means that `DXS(...)` and `XC(...)` include `ac(...)` in their definitions to access `Dxs` and `xc` respectively, but `DLONC(...)` and `GLATV(...)` do not include `ac(...)`.

The values `DLONC(i,j)` and `GLATV(i,j)` are obtained by calling `get_linspace_val(...)`. Longitudes and planetographic latitudes are generated on demand at the invocation of the needed macros. In fact, in the core of `init_coords_spheroid(...)`, a grid of uniformly distributed longitudes and planetographic latitudes generated with `init_coords_ortho(...)`—which makes use of the inexpensive `get_linspace_val(...)`—is invoked before generating the coordinates on the surface. By calling this inexpensive function on demand, storing the angular distances of every point in memory can be avoided. Also, generating the values of the halo becomes way easier: there is no need to exchange information between processors to fill the halos with the use of `get_linspace_val(...)`, as the function can distribute points at indices $i, j < 1$.

The initialization of the coordinates is complex, but the reader should remember that in the creation of the spatial coordinates of the spheroid, the longitudes and planetographic latitudes have been uniformly distributed first. Functions that distribute points uniformly along a linear distance are very computationally inexpensive so it does not make sense to store longitude and latitude information for each of the points in memory. Further information on the creation of the coordinates can be found in the source code of `init_coords_spheroid(...)`, found in `sw_coords.c`. It should be said that a lot of effort was put into creating a parallel cumulative sum, which is now part of `sppde-extensions.c` and is used at the definition of `init_coords_spheroid(...)`.

The initialization of physical variables such as u, v, η or h_B is currently done with `setzero_scaf(...)`, but this can be easily changed by combining `forall(...)` and `U(...)`, `V(...)`, `ETA(...)` or `HB(...)`, or by using `generate_scaf(...)`.

The values of the zonal wind at every planetographic latitude are also computed here and stored in a `malloc(...)`-initialized array size the length of the subdomain in the y direction.

Solver main loop

While general idea of the iterative process of the main loop has already been described in chapter 3, there are some steps that appear in the current Shallow Worlds implementation that do not appear in Figure 3.1. For example, before the start of the main loop of Shallow Worlds, a general `halo_update(...)` of the variables is carried out to ensure that the processors have the values to solve their subdomain.

The first thing that takes place after the time condition in Figure 3.1 once the `halo_update(...)` ends is a call to `save_for_paraview(...)`, which writes the current state of the shallow world `sw` in a file together with other simulation parameters.

Next follows a call to `add_vortex(...)`, which creates a perturbation of a given radius at a specified location of the domain, which can be either a plane or a spheroid. The currently implemented formula to determine the area of influence of the vortex on a spheroid is meant for spheres, although tests have been done with `get_vincenty(...)` to achieve higher precision in the computation of geodetic distances on spheroids. Despite the efforts, these have not been successful so far. The vortex must be inside the solver loop and the more time steps `add_vortex(...)` is called in, the bigger the perturbation. If the vortex is introduced during very few time steps, the perturbation will not develop, as gravity will make it collapse together with the presence of zonal winds. Moreover, if a very big perturbation was introduced before the main loop, undesired gravity waves would appear and so as to avoid drop-like perturbations, these cannot be created as impulses with notable amplitudes before the loop. Further information on the vortex can be found in the work of Arnau Sabatés[8].

The numerical integration of the equations happens after introducing the vortex and the result takes into account the introduced perturbation. The η field that `solve_cons_h(...)` sees is not the one initialized in `create_world(...)`, but the one modified by `add_vortex(...)`.

Once η, u and v have been computed, the boundary conditions have to be imposed just after they are integrated. `halo_update(...)` takes into account if the domain is periodic and by calling `do_channel(...)` in the case of `sw.probdef` being `PLANET`, two walls are created along the boundaries parallel to the x axis. The boundary conditions have been explained in section 3.7.

At the end of the loop, the simulation time has to be updated by using $t_{\text{sim}} = t^{n+1} = t^n + \Delta t$. The time condition will use this calculated value.

Other functions can be added where needed in the core of the loop. A test function which simulates rain by adding perturbations to the η field in random positions, `add_droplets(...)`, is commented out in the

loop. Another function that is present in the loop is `compute_Dt(...)`, which returns the maximum Δt allowed in the time step. This value is printed in the output of Shallow Worlds and smaller than user-specified Δt , the program stops.

Termination

In `destroy_world(...)`, the memory spaces that were allocated with `malloc(...)` or `calloc(...)` are freed.

5.4.4 The system of primitive coordinates

The core of the solver does assume any coordinate system and therefore it could probably be used for any kind of structured grids, as long as the user can provide the information required, from the variable sin Figure 5.8 to he contribution of the geometrical factors in the equations (see the source code of e.g. `solve_adv_u(...)`).

The current Shallow World version is set up to work with both the orthogonal and the spheroidal coordinate system. In this coordinate system and possibly for all non-flat coordinate systems (due to the properties of the structured grid),

- the centred point and the x-staggered S1 point have the same latitude (that ot the centred planetographic latitude),
- the centred point and the y-staggered S2 point have the same longitude (that of the centred longitude),
- the vertex point and the x-staggered S1 point have the same longitude (that of the staggered longitude), and
- the vertex point and the y-staggered S2 point have the same latitude (that of the staggered planetographic latitude).

This information is used in the Shallow Worlds header `sw.h` and in `read_zw(...)`. The reader has to remember that the solver assumes that, at least, the distribution of latitudes and the longitudes—whatever their reference is—is uniform. Currently, latitudes such as `GLATC(...)`, `GLATV(...)`... are defined in terms of `glatC(...)` or `glatV(...)`, to emphasize that in order to initialize the spheroid coordinate system, Shallow Worlds *distributes uniformly the planetographic or geodetic latitudes* and not the planetocentric or geocentric latitudes ("glatC" should be read as "graphic latitude for the centered point").

The solver does not assume that some points may have the same x or y surface coordinate[3]. The spatial coordinates are stored in memory and are generated through `init_coords_ortho(...)` or `init_coords_spheroid(...)`, or through a user-defined function that defines a new surface.

The spatial coordinates of the surface of the spheroid have been computed by performing a cumulative sum of the Δx and Δy, which is carried out by `cumsum_x(...)` and `cumsum_y(...)`, respectively. These functions were not required by the core of the solver, but they deserve a small reference in this report because it was no sooner than after two weeks of intensive work when the author came with the final definitions of the functions. Despite the fact that they are not currently used, the cumulative sum is an operation that may needed in post-processing and in some algorithms, so it should be left in the source code.

Generating the spatial coordinates of a spheroid is complex because its cross sections are elliptical and the perimeter of an ellipse cannot be computed with ease. In the definition of `init_coords_spheroid(...)` an accurate explanation is found. The coordinates generated with `init_coords_spheroid(...)` for a planet like Jupiter have a precision of 1-to-10% when the domain spans $\Delta\varphi = 40°$ and is divided into $n_y = 100$ points. It should not be said that the more points the greater the precision.

[3]This is related to the fact that the solver does not assume any coordinate system.

CHAPTER 6

Version history, test runs and performance

The current version of Shallow Worlds is the product of 5 months of work, which include theory sessions, the development of prototypes and validation tests. The current version of Shallow Worlds outputs practically (up to the 6th decimal place) the same results as the reference software developed in Matlab by Arnau Sabatés.

6.1 Version history

Enrique García Melendo is the original creator of Shallow Worlds (now the *legacy* Shallow Worlds). During his stay in the University of the Basque Country he developed and thoroughly tested a program to solve the shallow water equations. Written in C, the original version of Shallow Worlds did not use any of the advantages of MPI and thus, simulations of storms lasting some days required up to a week or more of computing. However, the simulations he carried out with the software were highly successful and closely predicted the behaviour of storms of Saturn. In fact, his simulations and findings were published[5;4] and are a motivation to the author. It is his experience with the original Shallow Worlds that is fundamental to the development of an improved version, together with the knowledge of C-programming of Manel Soria.

The development of the new Shallow Worlds started in the second half of March after several weeks of theory and a prototype written in Matlab. In Figure 6.1 the version history has been represented.

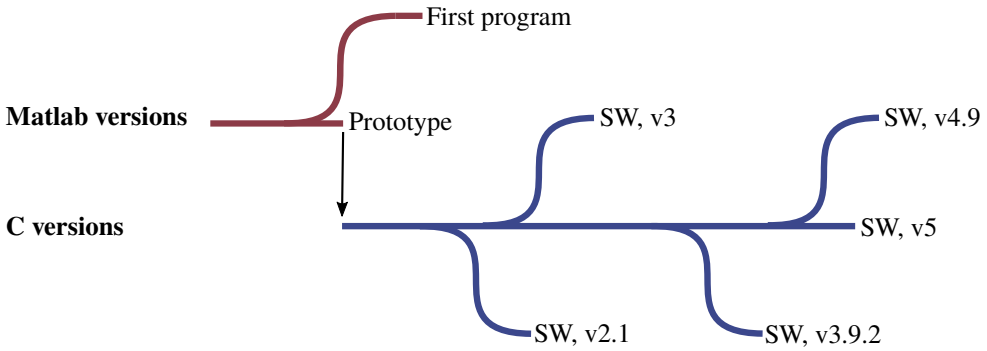


Figure 6.1: Version history

6.1.1 Prototype and First program

In order to understand the discretization and implementation of the shallow water equations in Shallow Worlds—which was to be programmed in C—a bare-bones version of the software was programmed in a

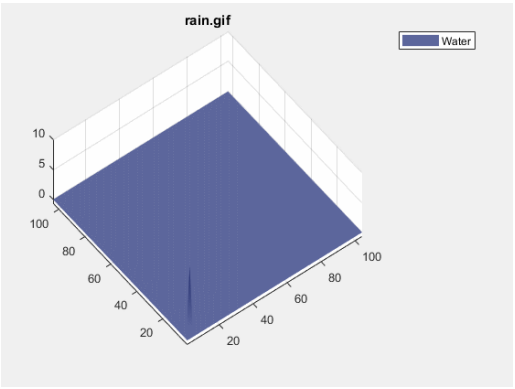
language that both Arnau Sabatés and the author are familiar with: Matlab.

Working with a proprietary language that has a clear and simple syntax, a powerful integrated development environment (IDE) and a number post-processing tools, eliminated the need to think of the memory management and the not-so-intuitive syntax features of C.

The goal was to independently develop a program that would compute the evolution of a drop (a perturbation at a point of the η field) in a periodic, flat domain. This program did not include the Coriolis force.

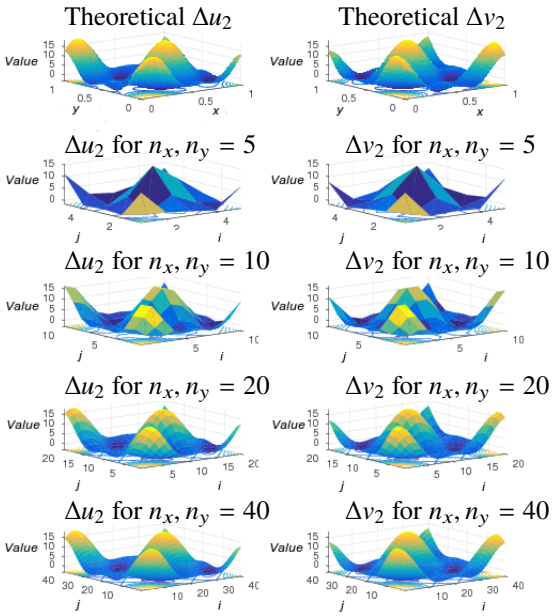
After a month of work, the author's Prototype was deemed as successful and it returned the same results as the software programmed by Arnau Sabatés. The whole source code of Prototype is available under request, as well as the First program's.

Further work was put in the Prototype, which lead to a more advanced version, codenamed First program. This program included post-processing tools hat produced the first animated simulations of rain (see the animation in Figure 6.2a), a feature also not included in Prototype.



(a) Animation of the rain simulation[a]

[a] Animations are only supported in Adobe Acrobat, so if the animation is not playing, it is available under request.



(b) MMS validation of the second part of the advection

Figure 6.2: Different outputs of the First program

First program—and consequently, Prototype—was also validated by means of the MMS and it was seen that the more resolution of the grid the closer the result was to the theoretical value (see Figure 6.2b). In the work of Arnau Sabatés the reader will find further information on the topic.

The development of the First program ended at this point in order to focus the efforts in the programming of Shallow Worlds. It should be said that First program was used to validate the first versions of the solver and that Arnau Sabatés continued improving his program while the Shallow Worlds's development started. As explained before, the reason he continued developing his solver was to provide the results that would be used to validate Shallow Worlds in the future.

6.1.2 Shallow Worlds milestones

The development of Shallow Worlds did not start immediately after the end of First program. During the first week, the author became familiar with MPI and `sppde` and designed the structure of the program together with Manel Soria.

At the end of March, the first version of the solver, `SW, v2.1`, was tested successful: First program and Shallow Worlds output the same results. The next milestone was to have basic visualization features, which was achieved in `SW, v3`. This version created files which could be post-processed by gnuplot. One of the animations produced with gnuplot is shown in Figure 6.3.

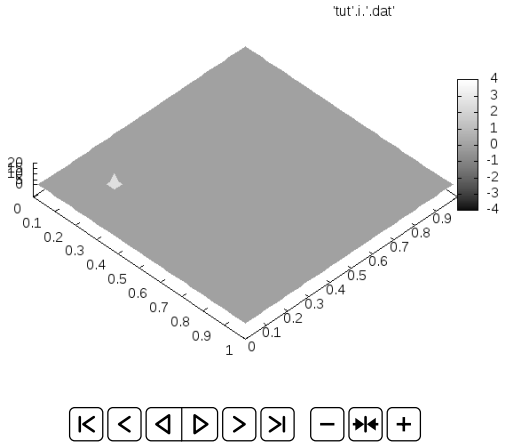


Figure 6.3: First Shallow Worlds animation[1]

In `SW, v3.9.2`, the program already produced files that could be processed with a filter in ParaView. This filter was created by Arnau Miró and modified by Manel Soria and the author. This version also included the spheroidal coordinate system as well as the Coriolis influence. The results were virtually identical to those of Arnau Sabatés. The author would like to point out that these minimal differences in the results can be attributed to the order in which the operations are carried out in low-level Matlab and C, as well as to the use of MPI/multiprocessors.

In `SW, v4.9`, a function that adds simple perturbations of the η fields that mimic a vortex (see `add_vortex(...)`) was added and successfully tested. Improvements in ParaView visualization were also carried out.

The current version series is `SW, v5` and is focused in improving the user experience. Information on the future Shallow Worlds improvements can found in chapter 9.

6.2 Performance tests and results

The Shallow Worlds uses MPI at its core and, consequently, a speed improvement over the original Shallow Worlds is expected. In this section, performance-related information of the new Shallow Worlds is presented.

6.2.1 Benchmarking tools

As part of the `sppde` library and included in `sppde_cr`, process timing functions are available for profiling. Currently, in the output of Shallow Worlds, the user will find information of the duration of every function in the core as presented in Listing 6.1. The minimum and the maximum computing time of the functions of all the time steps, as well as the sum of all the (per-time-step) times and their average, are computed by the `sppde` library.

Listing 6.1: Shallow Worlds benchmarking information

```
      nom o      N   mode    tmin       tmax       tsum       tavg    (s)
  time_step00  0     0 10    0 1.686597e-02 2.921789e-01 8.560455e-01 8.560455e-02
halo_update00  6     0 20    0 9.291172e-04 3.900695e-02 3.297789e-01 1.648895e-02
 compute_Dt00  1     0 10    0 5.793571e-05 2.420592e-02 3.334665e-02 3.334665e-03
solve_press00  8     0 10    0 1.981258e-04 5.193949e-03 7.028341e-03 7.028341e-04
  solve_adv00  7     0 10    0 5.741119e-04 1.025915e-03 6.741285e-03 6.741285e-04
solve_cons_h00  2     0 10    0 2.610683e-04 3.190041e-04 2.792120e-03 2.792120e-04
     newval00  5     0 20    0 2.861023e-06 8.797646e-05 5.517006e-04 2.758503e-05
 ABashforth00  3     0 20    0 1.907349e-06 9.059906e-06 1.065731e-04 5.328655e-06
  copy_Dvar00  4     0 20    0 2.861023e-06 5.960464e-06 9.274483e-05 4.637241e-06
     newvel00  9     0 10    0 5.006790e-06 1.001358e-05 6.675720e-05 6.675720e-06
```

[1]The speeding at the end of the animation is due to the use of variable Δt. Animations are only supported in Adobe Acrobat, so if the animation is not playing, it is available under request.

This information can be very useful to prove the speed improvements of parallel–MPI over sequential code. The following section is dedicated to the performance of Shallow Worlds.

6.2.2 Problem definition for benchmarking

In order to ensure that all test runs are carried out the same way, below in Listing 6.2 is a description of the simulation parameters.

Benchmarking simulation parameters

The simulation is of a storm similar to the Great Red Spot in Jupiter (see Listing 6.2). Therefore, `SW->probdef=PLANET` must be set to use the periodic channel boundary conditions, and set the domain according to $\lambda_0 = -60°$, $\lambda_1 = 60°$, $\varphi_g = -40°$ and $\varphi_g = 0°$.

Listing 6.2: Benchmarking Jupiter parameters

```
SW->Omega=1.76e-4; // Angular velocity
SW->rE=71492000.0; // Equatorial radius
SW->rP=66852000.0; // Polar radius
SW->mass=1.898e27; // Spheroid average mass
SW->dens=1.3; // Density of the atmosphere
```

All the variables u, v, η and h_B have to be set to zero, except for the zonal winds, which are specified in a file that can be delivered under request. In the core of the function, a vortex (Gaussian-shaped, amplitude of 100) has to be generated every 20 iterations, like in Listing 6.3, and writing on files must be done every 10 time steps, as shown in Listing 6.4.
The simulation has to last 50 time steps[a] of $\Delta t = 60$ s each and, in order to be aware of the progress of the solver, `info` should be set to 1.

Listing 6.3: Benchmarking vortex

```
if(n%20==1 && n<2500) {
        add_vortex((SW->lon0+SW->lon1)/2.0, (SW->glat0+SW->glat1)/2.0, 4e6, SW);
            // Longitude and latitude
}
```

Listing 6.4: Benchmarking save

```
if(n%10==1 || n==N) { // Save every 10 time steps
        save_for_paraview("/tmp/test", t, frame_num, SW);
        frame_num++;
}
```

[a] Future tests should not be 50 time steps long. Instead, the author suggests doing a test run of at least 1000 steps. Benchmarks should never start with different initial conditions as comparing the performance between runs becomes very difficult.

6.2.3 Test runs and results

At the time of writing, Shallow Worlds has not been tested on an adequate infrastructure such as clusters, but on general purpose computers. The results shown below have been obtained from different runs of the solver on the author's laptop (see the specifications in Listing 6.5).

Listing 6.5: Computer specifications

```
Processor: AMD A4-5000 APU
Memory: 5,3 GiB
Operating system: Ubuntu 16.04 LTS
```

Due to these characteristics, the data shown below is not representative of the potential of MPI, but interesting information can be extracted.

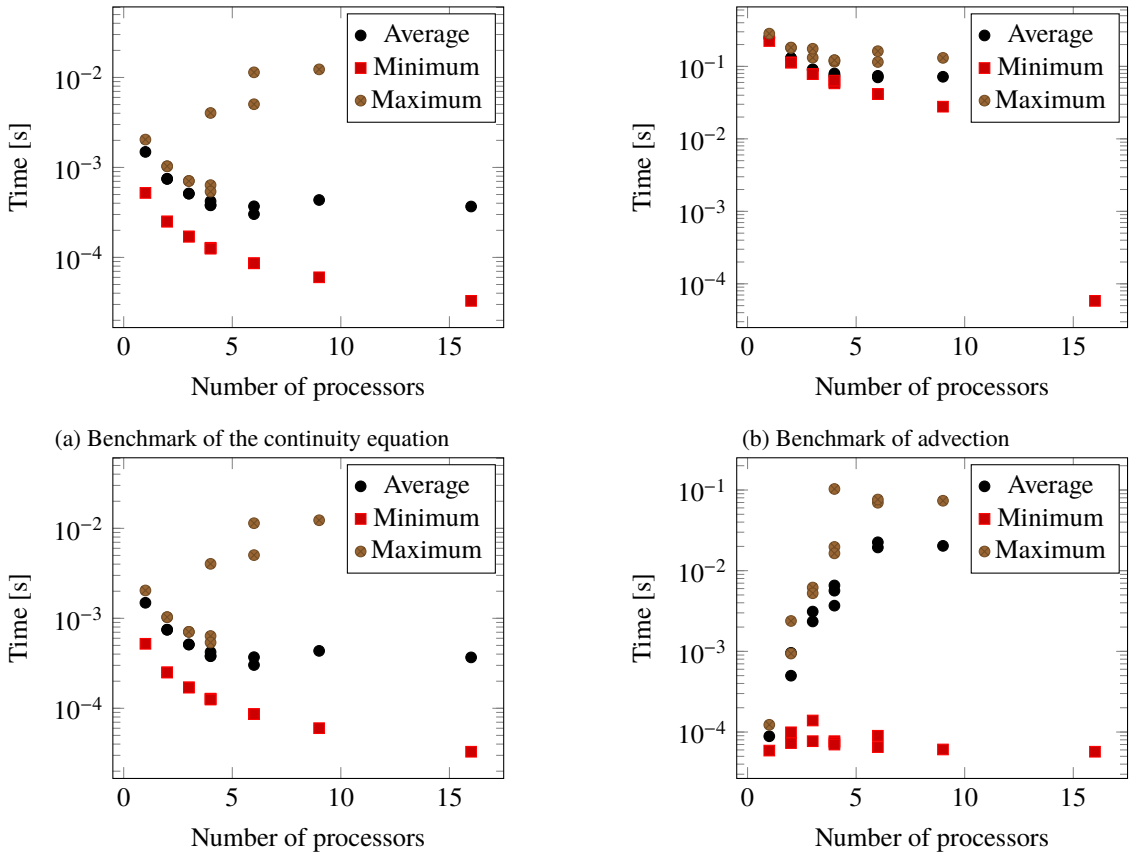
In the case of the advection and the continuity equation, the benchmark clearly shows how, by increasing the number of processors, the duration of the process is reduced. However, in Figure 6.4a and Figure 6.4b,

a plateau seems to be achieved. This can be attributed to the fact that the laptop cannot offer more than four [virtual] cores and therefore, splitting the tasks into different processes becomes inefficient as they have to be queued. The same behaviour can be seen in the call of `do_AdamsBashforth(...)` in Figure 6.4c and the same reasoning applies.

As for Figure 6.4d, a totally different tendency is observed. In this case, the average time increases with the number of processors. This can be attributed to the nature of the function, which is devoted to the communication between processors. Therefore, the more processors, the more complex and lengthy the call to `halo_update(...)` becomes. Also, the more coupled the cores are, the faster the operation is.

The interesting point is that, in the long run, it is worthy to use supercomputers to divide the domain into an optimal number of subdomains. Despite the halo update operation becoming more expensive with the number of processors, complex mathematical operations become much faster as they have to be applied in smaller subdomains.

A run on a cluster of computers specially optimized for numerical computation was scheduled during the course of this project. Because of time requirements this has been postponed for the presentation.



(a) Benchmark of the continuity equation

(b) Benchmark of advection

(c) Benchmark of Adams–Bashforth

(d) Benchmark of halo update

Figure 6.4: Benchmark results

CHAPTER 7

File input and output and basic post-processing

Because Shallow World is in its early stages, only very basic Input/Output (IO) is currently implemented. With the use of different tools, ASCII text-based files can be read and processed in ParaView, an industry standard for data visualization.

7.1 Reading and writing from and to files

The `sppde` library developed by Manel Soria includes file parser to process data stored in files, which is defined in `sppde_parser.h`, `sppde_parser.c`. The parser requires a small tool, written by Lewis Van Winkle, that evaluates math expressions that is contained in two files `sppde_tinyexpr.h` and `sppde_tinyexpr.c`,

The main use of a parser in a solver is to initialize simulation parameters and variables without without having to recompile the code. This means that the same binaries/executable files can be used to simulate different situations, which is extremely practical for the user. Moreover, if the structure of the file where the results are stored periodically matches the tool's requirements, any simulation can be started from the last results. Electricity cuts or computation errors which cause programs to stop are not unusual in some infrastructures, and having the assurance that the simulation's progress has not been lost is reassuring to the user.

Reading data is a crucial feature in Shallow Worlds that has not still been implemented as expected due to the ending of this FYP. The goal is to implement input and output of binary files instead of the current ASCII text-based files.

7.1.1 Reading zonal wind files

Currently, the only use of the file parser in Shallow Worlds is in the reading of the zonal wind through `read_zw(...)`. The file that contains the latitudes and the corresponding wind speed (relative to the planet's rotation) has the structure presented in Listing 7.1. In order to process the file, the first value has to be the number of latitudes/zonal wind readings (number of rows) and the second value, the number of columns. This is not a requirement imposed by `read_zw(...)` per se, but because this function has a call to `p_alloc_gettable(...)`—of the `sppde` parser—in its core.

Zonal wind file example

The zonal wind file is constituted by a column of latitudes and a column of the wind speed readings. This information can be presented in a `csv` file.

Listing 7.1: Structure of the zonal wind file

```
2915
2
-81.58 6.79
-78.93 -7.84
-78.89 -4.29
-78.56 -3.31
-78.05 -4.61
... ...
```

`read_zw(...)` fills the positions of `sw.zw` with the linearly interpolated values at the latitudes of the nodes of the mesh by using `linterp(...)` or `linterp_vec(...)`. The reader should know that the length of `sw.zw` is that of the subdomain plus the halo, so the processor only knows of the values it needs.

7.1.2 Writing the results in files

When Shallow Worlds is executed, every processor generates a log file that is stored, by default, in `/tmp/` under the name e.g. `stdout00.txt`. This file contains information of the time steps, partial results and debugging messages that the user may find of use but in any case should it be used for postprocessing.

The files that are used for post-processing are currently generated by the root processor, which in Shallow Worlds is the processor with rank 0 or the one at the bottom left of the processor grid. When `save_for_paraview(...)` is called, the root processor gathers the results of the other processors and writes them in a file with a particular structure. This file contains information of u (without zonal wind), v, w (the vertical velocity, currently not used in Shallow Worlds), η and Π_s (the potential vorticity), and can be read directly from Paraview (see section 7.2).

In section 7.2, the reader will find information of the structure of the file and its ParaView parser.

7.2 The SWreader: a Python script for ParaView

The order in which the variables are stored in the post-processing file has not been chosen randomly but imposed by a script that Arnau Miró originally developed. When this Python script is executed in ParaView, a structured grid is generated in the viewer from the data stored in the post-processing files, as well as in another file named `index.txt`.

The `index.txt` file is created in `/tmp/` by executing `./create_index /tmp` after Shallow Worlds. This new file contains basic information of the post-processing files and also the time mark of every file.

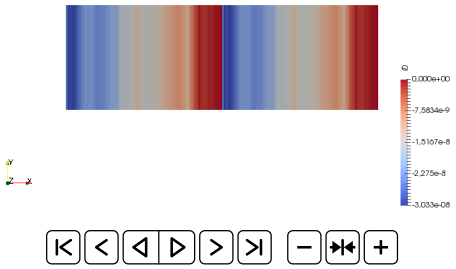
The program be loaded from the plug-in management window in the `xml` format under the name of SWreader. Once loaded, it can be used by clicking on `SWreader` in the Sources menu and specifying the location of the files.

Using this script may lead to ParaView failing in displaying the data. When a large number (> 10000) of post-processing files want to be plotted, ParaView 5.1 has been shown to freeze when the frame number is around half the number of imported files (depending on the resolution of the grid). While this is an error of ParaView, some effort has been put into overcoming the problem with SWreader but at the time of writing, the tool has not been validated.

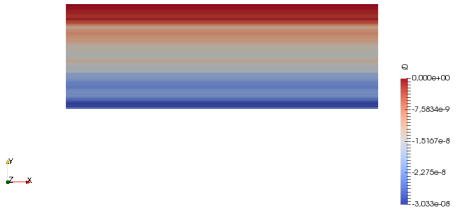
If the data has been imported correctly, the user sees the following in Figure 7.2a. From the dataset dropdown menu, the variables u, v, w, η and Π_s can be plotted and will appear as in Figure 7.2b. In chapter 8, the reader will find pictures and animations of different test simulations.

For multilayer simulations, where w may be needed in the output, the SWreader has not been shown to work as expected and further tests are required. This is due to the methodology used to amend the error present in the original SWreader. When the script was first used for rectangular domains the values seemed to be repeated along the x direction (see Figure 7.1a). The author saw that for plots of the domain with a

resolution Aspect Ratio (AR) $n_y/n_x \neq \mathbb{N}$, the data seemed to be shifted at every row, and therefore, an error in the loading of the data in the linear memory space (see Figure 5.6) was happening.



(a) Incorrect plots of Π_s in ParaView[1]



(b) Plot of Π_s in ParaView after the SWreader fix.

Figure 7.1: SWreader output

In the original script, the data was incorrectly passed to the structured grid creator, which only accepts one dimensional arrays. It was concluded that the matrices were wrongly "flattened" and passed to the creator.

```
data.ravel(order='F')
```

```
data.transpose(1,0,2,3).ravel(
    order='C')
```

By permutating the dimensions of the four dimensional arrays this was fixed.
2D simulations currently produced by Shallow Worlds are well supported if they are not long enough.

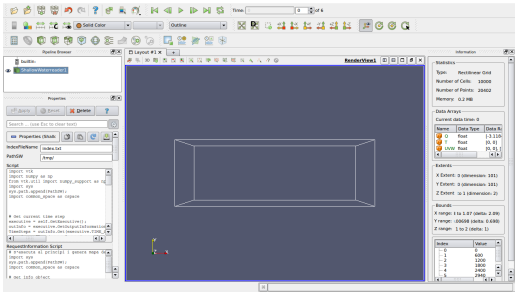
7.3 Exporting data to the csv

The `csv` file is understood by most of plotting tools and generating these files is quite easy. Shallow Worlds contains a very basic function that returns a *transposed* `csv` file. By calling, `save_tr_csv(...)`, the user will have coordinate data for every point of the domain, and of u, v, w, η and Π_s.

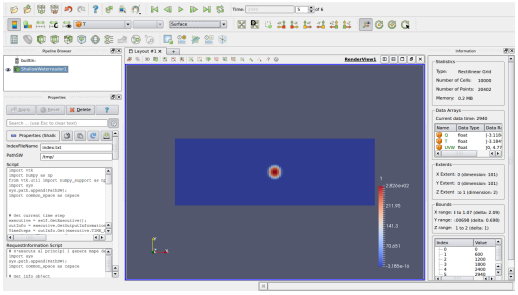
The file can be saved as a normal `csv` if `csvtool transpose input.csv > output.csv` is used on the file produced by Shallow Worlds.

This tool was created to test the visualization capabilities of ParaView. Below in Figure 7.2 are screenshots of ParaView usage with Shallow Worlds data. The goal is to deliver an animation of a storm in ellipsoidal coordinates in the presentation of this FYP.

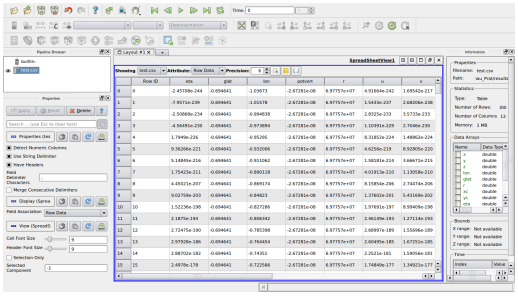
[1]The animation shows plots of n_y=80 and n_x = 160, 161, 162, 163, 166, 170, 180, 190, 200, 210, 220, 230, 234, 237, 238, 239, 240 of the benchmarking problem. Animations are only supported in Adobe Acrobat, so if the animation is not playing, it is available under request.



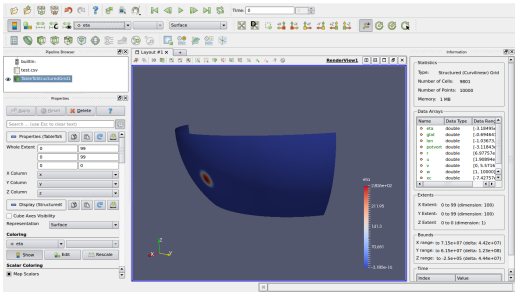
(a) ParaView screenshot after importing the data



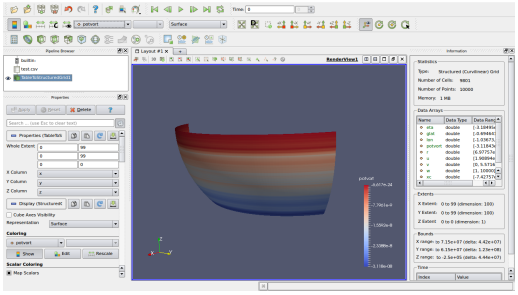
(b) ParaView plot of the step 41 of 50 of a simulation test



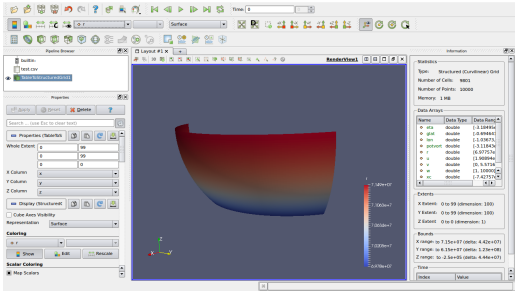
(c) Imported `csv` data in ParaView



(d) Plot of η in ellipsoidal coordinates in ParaView



(e) Plot of Π_s in ellipsoidal coordinates in ParaView



(f) Plot of r of the spheroid in ParaView

Figure 7.2: ParaView screenshots

CHAPTER 8

A compilation of simulations

In this chapter, the results of unrealistic simulations are presented and briefly commented. The author encourages the reader to study the work of Arnau Sabatés in order to follow the comments and make their own conclusions. With the information presented in this thesis and Arnau Sabatés's[8], the behaviour of the phenomena simulated in this chapter can be explained in physical terms.

8.1 Analysis of a demonstration case

In Figure 8.1 a plot of the potential vorticity Π_s can be seen. The first thing the reader should notice is the orientation of the axes: the z axis is pointing at the reader, and the y axis upwards. The presence of colour bands can be attributed to the zonal wind, which varies with the latitude (y axis). Because the Coriolis parameter increases with the latitude (see (2.8)) and the potential vorticity depends on the Coriolis parameter (see (2.39)), in most cases Π_s will also increase with the latitude. This can be used to check if the orientation of the plot is correct.

Another way to check the plot's orientation is by looking at the rotation of the perturbation. Because the vortex in Figure 8.1 is located in the south hemisphere of Jupiter, it rotates anticlockwise.

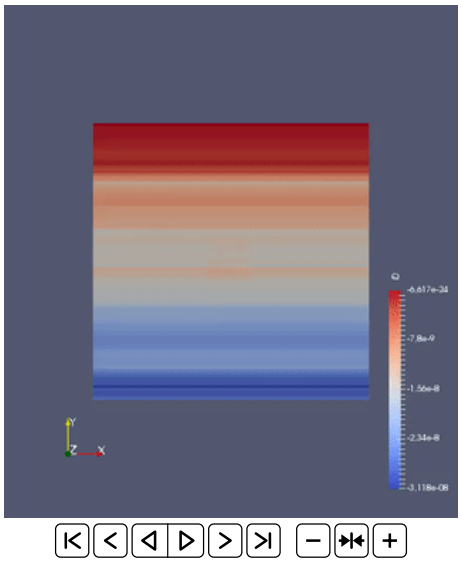


Figure 8.1: Animation of a perturbation at the latitude of the Great Red Spot[1]

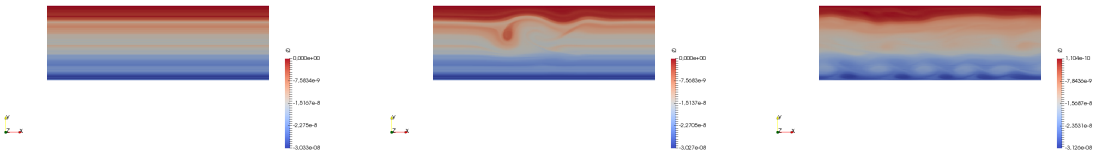
While this animation is not a simulation of a real situation, it can also be used to verify the periodic BC. As it can be seen, the storm disappears from the left side and appears in the right, as though the domain was a cylinder. This is the expected behaviour.

8.2 Presentation of a 100000 time step simulation

The longest simulation carried out by Shallow Worlds to this day is that of a simple version of the Great Red Spot of Jupiter. The goal of this execution was to verify that the program did not fail in long executions and with rectangular domains.

The solver solved a region of the atmosphere enclosed in $\Delta\lambda = 40$ and $\Delta\varphi = 120°$ with a resolution of 240-by-80 points. After 100000 time steps of 60 seconds each, the author expected to have a simulation of the life of a proto-vortex.

However, two errors were encountered when the results were visualized in ParaView. The first was caused by a mistake in SWreader (see section 7.2) and it was quickly fixed. The other error has been constantly reported by users of versions 5.1 to 5.4 and is due to memory management in ParaView. When large datasets are imported all together, the software fails to plot the data as expected. In this case, despite having very low resolution, a large number of time steps were available, which added up to $240 \cdot 80 \cdot 10000 = 192 \cdot 10^6$ data points, where 10000 is the number of files that contain results (i.e. frames of the animation). Such number of points could not be plotted on the author's computer and consequently, the animation of the results remained frozen for the most part of the steps. Only the first ≈ 1000 frames were correctly displayed, and from frames 1000 to 9999, ParaView showed a still image. When the last frame was achieved, the tool incomprehensibly displayed the [seemingly] correct data. In Figure 8.2 the reader will find the mentioned frames. A 40 second animation is available.



(a) First frame of the simulation (b) Frozen state (c) Last frame of the simulation

Figure 8.2: Results of the 100000-frame simulation

[1] Animations are only supported in Adobe Acrobat, so if the animation is not playing, it is available at request.

CHAPTER 9

The future of the solver

Shallow Worlds does not end with the end of this FYP as there are many actions that could be carried out in order to improve the solver's usability and performance. The author has divided these according to the expected delivery date: if the improvements will be introduced in the presentation of the FYP, they are classified as *immediate*. If not, they have been called *long term*.

9.1 Immediate actions

Below is a compilation of the immediate actions that will be performed before the presentation:

- The author's goal is to validate Shallow Worlds against its predecessor, the legacy Shallow Worlds. The methodology that will be used is the same as the employed with the prototype built by Arnau Sabatés: if the results both solvers output are close enough, the new Shallow Worlds will be validated.
- Shallow Worlds is delivered with `generate_scaf(...)`, which can be used to generate scalar fields according to a mathematical function. This function is pending validation.
- At the time of writing, all the simulations have been carried out for $h_B = 0$. Even though that the planet topography is implemented in Shallow Worlds it has not been tested yet. Tests with user-generated relief will be done.
- A feature that will be implemented soon is the reading of the simulation parameters from a file. This will greatly improve the ease of use as the code will not have to be recompiled every time a simulation parameter is changed.
- Even though the author of this thesis has put a lot of effort in the presentation of the source code, a lot of comments are still missing. The documentation should be improved as soon as possible.
- The software has not been tested on different infrastructures, even though it has the potential to perform decently even on large clusters of computers. A scalability test could not be done during the development of the solver because of the schedule. The author would like to run this program in a cluster of computers to verify the correct implementation of MPI.
- The research group has satellite data of different phenomena. The aim is to use this data as the initial conditions of a simulation and prove that Shallow Worlds closely represents the reality.

9.2 Long term actions

A list of actions that are still not scheduled to be executed in the future is presented below:

- The current generation of the coordinates is not very clear. Improvements in this directions are a must. Also, this solver should be tested with domains not currently supported.
- Binary file IO is required when the solver is put under constant use and a large quantity of files are generated. This feature should be implemented as soon as possible.
- The program should follow the trends and therefore output data in commonly used file formats. NetCFD and VTK support could be implemented.
- The current version of Shallow Worlds does not support variations in z of any of the variables. If the layers feature is added, stratified atmospheres could be represented with accuracy.
- The long term goal is to achieve a global circulation model. This solver could be used to set the bases for a future program.

Part III

Budget and environmental impact

CHAPTER 1

Budget

In this chapter, a compilation of all the costs and expenses related to the realization of this FYP is presented. The figures in Table 1.1 are approximations.

The costs of this project can be mainly attributed to personnel costs and the powering of computers (see Table 1.1). The price of a Matlab license is also included in the budget, even though that if the software had not been available, an open source alternative such as Julia or Python might have been used.

Table 1.1: Costs of the study

Concept	Amount	Unit cost	Attributed
Engineering work	630 h	16 $\frac{€}{h}$	10 080 €
Matlab license	1 u.	800 $\frac{€}{u.}$	800 €
Electricity consumption	65 W for 600 h	0.17 $\frac{€}{kW \cdot h}$	7 €
Computer depreciation	650 € per 60 mo.	11 $\frac{€}{mo.}$	65 €
Total cost	NaN	NaN	10 951 €

The author has assumed the hourly salary of a junior engineer to obtain a first approximation of *Engineering work* and that the depreciation of the value of the computer with time is linear.

At the time of writing, Shallow Worlds has not been thought of a viable source of income. If the development of the solver continues and it becomes a flagship of a department, the research could be sustained with funds and initiatives coming from academic organizations, enterprises, and governmental and intergovernmental institutions.

CHAPTER 2

Environmental impact

Different amounts of CO_2 have been emitted to the atmosphere during this FYP according to different estimates:

- According to the average of the 2015 Factor Mix published by the Industry Ministry of Spain[?], 4.68 kg of CO_2 have been emitted to the atmosphere. The emissions factor used in this case is 0.12 $\frac{kg}{kW \cdot h}$ of CO_2.
- This factor should not be used as it does not take into account the weight of each of the enterprises. A better approximation is 0.245 $\frac{kg}{kW \cdot h}$ of CO_2, which leads to an emitted amount of 9.56 kg of CO_2.
- A conservative estimate of the emissions is 15.60 kg of CO_2 for a factor of 0.4 $\frac{kg}{kW \cdot h}$ of CO_2.
- In the past, a factor of 0.649 $\frac{kg}{kW \cdot h}$ of CO_2 was used. In this case, the attributable to this FYP is 25.31 kg of CO_2.

No other environmental factors have been taken into account in this FYP.

As a side note, large clusters of computers such as supercomputers are very energy demanding and their environmental impact may be very significant.

Bibliography

[1] Adobe Systems Incorporated (2000). The HSB/HLS Color Model - Color Models - Technical Guides.

[2] Arakawa, A. and Lamb, V. R. (1977). Computational Design of the Basic Dynamical Processes of the UCLA General Circulation Model. In CHANG, J., editor, *General Circulation Models of the Atmosphere*, volume 17 of *Methods in Computational Physics: Advances in Research and Applications*, pages 173–265. Elsevier.

[3] Dowling, T. E., Fischer, A. S., Gierasch, P. J., Harrington, J., Lebeau, R. P., and Santori, C. M. (1998). The Explicit Planetary Isentropic-Coordinate (EPIC) Atmospheric Model. *Icarus*, 132(2):221–238.

[4] García-Melendo, E. and Sánchez-Lavega, A. (2017). Shallow water simulations of Saturn's giant storms at different latitudes. *Icarus*, 286:241–260.

[5] García-Melendo, E., Sánchez-Lavega, A., and Dowling, T. E. (2005). Jupiter's 24° N highest speed jet: Vertical structure deduced from nonlinear simulations of a large-amplitude natural disturbance. *Icarus*.

[6] Kämpf, J. (2009). *Ocean modelling for beginners: Using open-source software*. Springer-Verlag Berlin Heidelberg, 1 edition.

[7] Pedlosky, J. (1987). Geophysical Fluid Dynamics. *New York and Berlin, Springer-Verlag, 1982. 636 . . .*, page 742.

[8] Sabatés, A. (2018). *Analysis and study of a shallow water model code for applications to planetary atmospheres*. PhD thesis, Universitat Politècnica de Catalunya.

[9] Soria Guerrero, M. (2005). Parallel multigrid algorithms for computational fluid dynamics and heat transfer. *TDX (Tesis Doctorals en Xarxa)*.

[10] Soria Guerrero, M. (2017). Fundamentals of finite control volume spectroconsistent CFD.

[11] Thyng, K., Greene, C., Hetland, R., Zimmerle, H., and DiMarco, S. (2016). True Colors of Oceanography: Guidelines for Effective and Accurate Colormap Selection. *Oceanography*, 29(3):9–13.

[12] Vincenty, T. (1975). Direct and Inverse Solutions of Geodesics on the Ellipsoid With Application of Nested Equations. *Survey Review*, 23(176):88–93.

Acronyms

AB Adams–Bashforth method. 18

AR Aspect Ratio. 52

BC boundary condition. 16, 29, 30, 34, 39, 40, 55, 94, *Glossary:* BC

BSC Barcelona Supercomputing Center. 35

CV control volume. 19, 20, 22, 24, 42, *Glossary:* CV

FDM Finite Difference Method. 19, 24, 28, 39

FYP Final Year Project. 2, 4, 33, 50, 52, 56, 59, 60

HPC high performance computing. 35–37

IDE integrated development environment. 46, *Glossary:* IDE

IO Input/Output. 50, 57, *Glossary:* IO

MMS The Method of Manufactured Solutions. 33, 34, 46

MPI Message Passing Interface. 2, 5, 34, 35, 37, 40, 45–48, 56

NSE Navier–Stokes equations. 2, 7, 8, 35, *Glossary:* NSE

ODE Ordinary Differential Equation. 19

PDE Partial Differential Equation. 19, 63

TVD Total Variation Diminishing. 19, 21, 24, 26, 39, *Glossary:* TVD

Glossary

BC A set of values that must be satisfied by the solution of a boundary value problem.. 16

CV The region of the domain where the fluid equations are applied.. 19

deadlock A state where different programs are not progressing due to a coordination error or mistake has been encountered and they depend on one another for completion.. 37, 40

HPC The practice of using special computer architectures to achieve a greater computing power than the one offered by a typical desktop computer.. 35

IDE An application that facilitates the development of software.. 46

IO The input and output of the program.. 50

log file A file that records the events that occur.. 51

NSE The set of equations that describe the behaviour of the fluids.. 2

supercomputer A high performance computer compared to a general-purpose computer.. 35

TVD Related to the numerical solving of some PDEs, is a property of some schemes.. 19

Part IV

Reference guide for Shallow Worlds

APPENDIX A

First steps

In order to carry out a simulation in the current version of Shallow Worlds the following steps have to be taken. This is not the final solution. In fact, the goal is to be able to enter this data from a file before the presentation as the reader will see that it is very inconvenient.

1. Enter the parameters in `create_world(...)`:
   (a) Define `sw.probdef` to `DROP` or `PLANET`.
   (b) Set the domain limits. For case `DROP` specify the spatial coordinates as well as the angular. For the f-plane case set the start and end latitudes equal or almost equal.
   (c) Read the zonal wind file with `read_sw(...)`.
   (d)
2. Set the Δt to the desired value in `do_core_loop(...)`.
3. Compile the code with `./compile`.
4. Run the code with `mpirun -np 4 sw 100 100 2 2 1000`. This will perform a solve the domain with a resolution of 100-by-100 points, with 2 processors per side (in total, 4 processors, as specified in `-np 4`) for 1000 time steps.
5. Execute `./compile_check /tmp/`. `/tmp/` is the location where the files are saved by default.
6. Read the data in ParaView using the SWreader.
7. DONE!

APPENDIX B

Source code

B.1 Header

In `sw.h`, all the macros and prototypes are present. Here, the equivalences between e.g. equal latitudes of the different points of the cell are defined.

```
#define G_universal 6.67408e-11 // Universal gravity constant
#define PI (4.0*atan(1.0)) // Pi value dependant on math.h's atan(...) implementation
//#define PI M_PI // Fixed pi value as defined in math.h

// PROBLEM PROTOTYPES
```

Both `DROP` and `PLANET` are used as identifiers that select which equations to use in the body of the functions, such as in `create_world(...)`.

```
#define DROP 10
```

`DROP` revised on 01/06/2018 by Arnau Prat Gasull. Drop problem type identifier.

```
// #define FPLANE 11
```

`FPLANE` revised on 01/06/2018 by Arnau Prat Gasull. f-plane problem type identifier.

```
// #define BETAPLANE 12
```

`BETAPLANE` revised on 01/06/2018 by Arnau Prat Gasull. β-plane problem type identifier.

```
#define PLANET 13
```

`PLANET` revised on 01/06/2018 by Arnau Prat Gasull. Planet problem type identifier.

```

typedef struct {
```

`sw` revised on 01/06/2018 by Arnau Prat Gasull.

Description

Contains the information required to define a Shallow World.This is the structure upon the program is set up and is treated like an object (similarly to Object Oriented Programming). in a way, it contains the intrinsic properties of the Shallow World.

```
    int probdef; // Problem definition, see above
    int nx, ny; // Number of cells in the centered mesh ( there are nx+1 and ny+1
        division lines)
    double lon0,lon1; // Initial and final longitude
    double glat0,glat1; // Initial and final latitude
    double x0,x1; // Initial and final coordinates
    double y0,y1; // Initial and final coordinates
    map m; // Intrinsic properties of the domain
    map *M; // Pointer to m, to ease notation

```

```
23          // Scalar values
24          double alpha; // Value for hibrid schemes
25          double Omega; // Angular velocity
26          double rE,rP; // Equatorial and polar radii of the spheroid
27          double epsilon2; // Radius ratio of the spheroid
28          double mass; // Spheroid average mass
29          double dens; // Density of the atmosphere
30          double D; // Reference depth of the fluid layer
31
32          // Scalar fields
33          double *u,*v,*eta,*hB;
34          double *zw;
35          // double *lons,*lonc,*lonv,*glats,*glatc,*glatv;
36          double *xs,*xc,*xv,*ys,*yc,*yv;
37          double *Dxs,*Dxc,*Dxv,*Dys,*Dyc,*Dyv; // Distances between centered points and
                staggered points (centered, staggered, edge)
38          // double *xc,*xs,*yc,*ys; // Coordinates of the centered points and staggered
                points
39  } sw;
40
41  // ACCESS MACROS FOR THE SHALLOW WORLD VARIABLES
42
43  // Commonly used macros
44  #define MIN(a,b) (((a)<(b))?(a):(b))
45  #define MAX(a,b) (((a)>(b))?(a):(b))
46  #define POW2(a) ((a)*(a))
47  #define POW3(a) ((a)*(a)*(a))
48  #define deg2rad(a)  ((a)*(PI/180.0))
49  #define rad2deg(b)  ((b)*(180.0/PI))
50
51  // Spheroidal coordinates
52  #define lonS(i) (get_linspace_val(i,SW->lon0,SW->lon1,SW->nx))
```

`lonS(...)` revised on 01/06/2018 by Arnau Prat Gasull. Longitude of the staggered (x) point. Longitudes can only be staggered in x.

```
53  #define lonC(i) ((lonS(i)+lonS(i-1))/2.)
```

`lonC(...)` revised on 01/06/2018 by Arnau Prat Gasull. Longitude of the centered point. The centered point falls in the middle of staggered points

```
54  #define lonV(i) (lonS(i))
```

`lonV(...)` revised on 01/06/2018 by Arnau Prat Gasull. Longitude of the vertex point according to the geometry of the cell.

```
55  #define glatS(j) (get_linspace_val(j,SW->glat0,SW->glat1,SW->ny))
```

`glatS(...)` revised on 01/06/2018 by Arnau Prat Gasull. Graphic latitude of the staggered (y) point. Latitudes can only be staggered in y.

```
56  #define glatC(j) ((glatS(j)+glatS(j-1))/2.)
```

`glatC(...)` revised on 01/06/2018 by Arnau Prat Gasull. Graphic latitude of the centered point. The centered point falls in the middle of staggered points

```
57  #define glatV(j) (glatS(j))
```

`glatV(...)` revised on 01/06/2018 by Arnau Prat Gasull. Graphic latitude of the vertex point according to the geometry of the cell.

```
58
59  // Latitude and longitude matrices
60
```

The following equivalences have been defined so the solvers `solve_cons_h(...)`, `solve_adv_u(...)`, `solve_adv_v(...)`, `solve_pres_u(...)`, `solve_pres_v(...)`, `solve_Coriolis_u(...)`, `solve_Coriolis_v(...)`... are expressed without assuming any kind of geometry (except for rectangular cells).

```
61  #define LONS1(i,j) lonS(i)
```

`LONS1(...)` revised on 01/06/2018 by Arnau Prat Gasull. Longitude of a staggered S1 (x) point.

```
62  #define LONS2(i,j) lonC(i)
```

`LONS2(...)` revised on 01/06/2018 by Arnau Prat Gasull. Longitude of a staggered S2 (y) point.

```
63  #define LONC(i,j) lonC(i)
```

`LONC(...)` revised on 01/06/2018 by Arnau Prat Gasull. Longitude of a centered point.

```
#define LONV(i,j) lonV(i)
```
`LONV(...)` revised on 01/06/2018 by Arnau Prat Gasull. Longitude of a vertex point.

```
#define GLATS1(i,j) glatC(j)
```
`GLATS1(...)` revised on 01/06/2018 by Arnau Prat Gasull. Graphic latitude of a staggered S1 (x) point.

```
#define GLATS2(i,j) glatS(j)
```
`GLATS2(...)` revised on 01/06/2018 by Arnau Prat Gasull. Graphic latitude of a staggered S2 (y) point.

```
#define GLATC(i,j) glatC(j)
```
`GLATC(...)` revised on 01/06/2018 by Arnau Prat Gasull. Graphic latitude of a centered point.

```
#define GLATV(i,j) glatV(j)
```
`GLATV(...)` revised on 01/06/2018 by Arnau Prat Gasull. Graphic latitude of a vertex point.

```

// Coordinates of the centered points and staggered points

```

Even though a plane or a spheroid do not require to store the coordinates of each point in-memory (a vector for each axis would suffice), because the core of the solver has been developed to be as generalized as possible and to avoid expensive operations (in terms of computation) such as for an ellipsoid, a provisional decision to store the information of every point in-memory was made. A great deal of memory could be freed if the solver will only solve revolution objects.

```
#define DXS(i,j) ac(SW->Dxs,i,j,SW->M)
```
`DXS(...)` revised on 01/06/2018 by Arnau Prat Gasull. Distance on x between staggered S1 (x) points.

```
#define DXC(i,j) ac(SW->Dxc,i,j,SW->M)
```
`DXC(...)` revised on 01/06/2018 by Arnau Prat Gasull. Distance on x between centered points.

```
#define DXV(i,j) ac(SW->Dxv,i,j,SW->M)
```
`DXV(...)` revised on 01/06/2018 by Arnau Prat Gasull. Distance on x between vertex points.

```
#define DYS(i,j) ac(SW->Dys,i,j,SW->M)
```
`DYS(...)` revised on 01/06/2018 by Arnau Prat Gasull. Distance on y between staggered S2 (y) points.

```
#define DYC(i,j) ac(SW->Dyc,i,j,SW->M)
```
`DYC(...)` revised on 01/06/2018 by Arnau Prat Gasull. Distance on y between centered points.

```
#define DYV(i,j) ac(SW->Dyv,i,j,SW->M)
```
`DYV(...)` revised on 01/06/2018 by Arnau Prat Gasull. Distance on y between vertex points.

```
#define XS(i,j) ac(SW->xs,i,j,SW->M)
```
`XS(...)` revised on 01/06/2018 by Arnau Prat Gasull. x coordinates of a staggered S1 (x) point.

```
#define XC(i,j) ac(SW->xc,i,j,SW->M)
```
`XC(...)` revised on 01/06/2018 by Arnau Prat Gasull. x coordinates of a centered point.

```
#define XV(i,j) ac(SW->xv,i,j,SW->M)
```
`XV(...)` revised on 01/06/2018 by Arnau Prat Gasull. x coordinates of a vertex point.

```
#define YS(i,j) ac(SW->ys,i,j,SW->M)
```
`YS(...)` revised on 01/06/2018 by Arnau Prat Gasull. y coordinates of a staggered S2 (y) point.

```
#define YC(i,j) ac(SW->yc,i,j,SW->M)
```
`YC(...)` revised on 01/06/2018 by Arnau Prat Gasull. y coordinates of a centered point.

```
#define YV(i,j) ac(SW->yv,i,j,SW->M)
```
`YV(...)` revised on 01/06/2018 by Arnau Prat Gasull. y coordinates of a vertex point.

```

// #define ds_dclat(clat) ((SW->rP)*sqrt(1-(1-(SW->epsilon2))*POW2(sin(clat)))) // Same as rM
#define clat2glat(clat) atan((SW->epsilon2)*tan(clat))
```
`clat2glat(...)` revised on 01/06/2018 by Arnau Prat Gasull. Planetocentric latitude to planetographic latitude operator.

```
#define glat2clat(glat) atan(1.0/(SW->epsilon2)*tan(glat))
```
`glat2clat(...)` revised on 01/06/2018 by Arnau Prat Gasull. Planetographic latitude to planetocentric latitude operator.

```
#define fEP (((SW->rE)-(SW->rP))/(SW->rP))
```
`fEP` revised on 01/06/2018 by Arnau Prat Gasull. Second flattening of the ellipse.

```
#define r(clat) ((SW->rE)*(SW->rP)/sqrt(POW2(SW->rE*sin(clat))+POW2(SW->rP*cos(clat))))
```
`r(...)` revised on 01/06/2018 by Arnau Prat Gasull. Planetocentric radius of the spheroid.

```
// #define r(clat) ((SW->rE)*(1.0-fEP*POW2(sin(clat))))
```
`r_approx(...)` revised on 01/06/2018 by Arnau Prat Gasull. An approximation of `r(...)`.

```
#define rZ(glat) ((SW->rE)/sqrt(1.0+POW2(tan(glat))/(SW->epsilon2)))
```
`rZ(...)` revised on 01/06/2018 by Arnau Prat Gasull. Zonal radius of the spheroid.

```
#define rM(glat) ((SW->rE)/(SW->epsilon2)*POW3(rZ(glat)/((SW->rE)*cos(glat))))
```
`rM(...)` revised on 01/06/2018 by Arnau Prat Gasull. Meridional radius of the spheroid.

```

// Gravity definition
#define g0(clat) (G_universal*(SW->mass)/POW2(r(clat)))
```
`g0(...)` revised on 01/06/2018 by Arnau Prat Gasull. Reference gravity of the spheroid.

```
#define g_eff(clat) (g0(clat)-POW2((SW->Omega))*r(clat)*cos(clat))
```
`g_eff(...)` revised on 01/06/2018 by Arnau Prat Gasull. Effective gravity of the rotating spheroid.

```

// Gravitiy matrices
#define GS1(i,j) g_eff(glat2clat(GLATS1(i,j)))
```
`GS1(...)` revised on 01/06/2018 by Arnau Prat Gasull. Effective gravity at a staggered S1 (x) point.

```
#define GS2(i,j) g_eff(glat2clat(GLATS2(i,j)))
```
`GS2(...)` revised on 01/06/2018 by Arnau Prat Gasull. Effective gravity at a staggered S2 (y) point.

```
#define GC(i,j) g_eff(glat2clat(GLATC(i,j)))
```
`GC(...)` revised on 01/06/2018 by Arnau Prat Gasull. Effective gravity at a centered point.

```
#define GV(i,j) g_eff(glat2clat(GLATV(i,j)))
```
`GV(...)` revised on 01/06/2018 by Arnau Prat Gasull. Effective gravity at a vertex point.

```

// Definition of the Coriolis parameter
#define f(glat) (2.0*(SW->Omega)*sin(glat))
```
`f(...)` revised on 01/06/2018 by Arnau Prat Gasull. Coriolis parameter.

```

// Matrices with the Coriolis parameter
#define FS1(i,j) f(GLATS1(i,j))
```
`FS1(...)` revised on 01/06/2018 by Arnau Prat Gasull. Coriolis parameter at a staggered S1 (x) point.

```
#define FS2(i,j) f(GLATS2(i,j))
```
`FS2(...)` revised on 01/06/2018 by Arnau Prat Gasull. Coriolis parameter at a staggered S2 (y) point.

```
#define FC(i,j) f(GLATC(i,j))
```
`FC(...)` revised on 01/06/2018 by Arnau Prat Gasull. Coriolis parameter at a centered point.

```
#define FV(i,j) f(GLATV(i,j))
```
`FV(...)` revised on 01/06/2018 by Arnau Prat Gasull. Coriolis parameter at a staggered S2 (y) point.

```

// Main matrices
#define ETA(i,j) ac(SW->eta,i,j,SW->M)
```
`ETA(...)` revised on 01/06/2018 by Arnau Prat Gasull. The surface perturbation at a centered point.

```
#define U(i,j) ac(SW->u,i,j,SW->M)
```
`U(...)` revised on 01/06/2018 by Arnau Prat Gasull. The horizontal velocity at a staggered S1 (x) point.

```
#define V(i,j) ac(SW->v,i,j,SW->M)
```
`V(...)` revised on 01/06/2018 by Arnau Prat Gasull. The vertical velocity at a staggered S2 (y) point.

```
#define HB(i,j) ac(SW->hB,i,j,SW->M)
```
`HB(...)` revised on 01/06/2018 by Arnau Prat Gasull. The surface relief at a centered point.

```
#define H(i,j) ((ETA(i,j))+(SW->D)-(HB(i,j)))
```
`H(...)` revised on 01/06/2018 by Arnau Prat Gasull. *Read only*. The layer depth at a centered point.

```
#define ZW(i,j) SW->zw[j-SW->M->l0[2]+SW->M->sh]
```
`ZW(...)` revised on 01/06/2018 by Arnau Prat Gasull. Zonal wind at a staggered S1 (x) point.

```
#define UZW(i,j) (U(i,j)+ZW(i,j))
```
`UZW(...)` revised on 01/06/2018 by Arnau Prat Gasull. *Read only*. The sum of the zonal wind `ZW(...)` and the horizontal velocity `U(...)` at a staggered S1 (x) point.

```

// POSTPROCESSING

#define UC(i,j) ((U(i-1,j) + U(i,j)) / 2.)
```
`UC(...)` revised on 01/06/2018 by Arnau Prat Gasull. The horizontal velocity at a centered point.

```
#define VC(i,j) ((ac(SW->v,i,j-1,SW->M) + ac(SW->v,i,j,SW->M)) / 2.)
```
`VC(...)` revised on 01/06/2018 by Arnau Prat Gasull. The vertical velocity at a centered point.

```
#define UZWC(i,j) ((UZW(i-1,j) + UZW(i,j)) / 2.)
```
`UZWC(...)` revised on 01/06/2018 by Arnau Prat Gasull. *Read only*. The sum of the zonal wind `ZW(...)` and the horizontal velocity `U(...)` at a centered point.

```
#define DVDXV(i,j) ((V(i+1,j)-V(i,j))/DXV(i,j))
#define DUZWDYV(i,j) ((UZW(i,j+1)-UZW(i,j))/DYC(i,j+1))
#define VORT(i,j) (DVDXV(i,j) - DUZWDYV(i,j))
```
`VORT(...)` revised on 01/06/2018 by Arnau Prat Gasull. The vorticity at a vertex point.

```
#define PVORT(i,j) ((VORT(i,j)+FV(i,j))/(0.25*(H(i,j)+H(i+1,j)+H(i,j+1)+H(i+1,j+1))))
```
`PVORT(...)` revised on 01/06/2018 by Arnau Prat Gasull. The potential vorticity at a vertex point.

```
#define PVORTC(i,j) (PVORT(i,j)+PVORT(i-1,j)+PVORT(i,j-1)+PVORT(i-1,j-1))
```
`PVORTC(...)` revised on 01/06/2018 by Arnau Prat Gasull. The potential vorticity at a centered point.

```

// PROTOTYPES

// Analytic functions
double sombrero(double x, double y, double t);
double cosine(double x, double y, double t);
double unitary(double x, double y, double t);

// Commonly used functions
double linterp_core(double x_i, double *x, double *y, int *i0, int *i1);
double linterp(double x_i, double *x, double *y, int n);
void linterp_vec(double *x_i, double *y_i, int n_i, double *x, double *y, int n);
void read_zw(char *fname, sw *SW);
void copy_matrix(double *matrixCopy, double *matrixInput, int copy_halo, map *M);
double get_vincenty(double lon0, double lat0, double lon1, double lat1, double a, double
    b);
double get_linspace_val(int i,double x0,double x1,int nx);
double get_arc_ellipse(double a1, double rP, double epsilon2);
double get_arc_circumf(double a1, double r);
void generate_scaf(double *scaf, double (*funcio)(double,double,double), int stgx, int
    stgy, double t, sw *SW); // Generate scaf from user defined analytic function
double generate_val(int i, int j, double (*funcio)(double,double,double), int stgx, int
    stgy, double t, sw *SW); // Similarly to generate scaf, generates a value

// Shallow world initialization functions
void init_coords_ortho(sw *SW);
void init_coords_spheroid(sw *SW);
void init_terrain(sw *SW);
void init_velocities(sw *SW);
void init_surface_perturbation(sw *SW);
void init_Coriolis(sw *SW);
void init_gravity(sw *SW);
void create_world(int nx,int ny,int npx,int npy,sw *SW);

// Core functions
void do_core_loop(int N, sw *SW);

// Calc functions
```

```
void solve_cons_h(double *Deta_n, double Dt, sw *SW);
void solve_adv_u(double *Du_adv, double Dt, sw *SW);
void solve_adv_v(double *Dv_adv, double Dt, sw *SW);
void solve_pres_u(double *Du_pres, double Dt, sw *SW);
void solve_pres_v(double *Dv_pres, double Dt, sw *SW);
void solve_Coriolis_u(double *Du_nAB, double Dt, sw *SW);
void solve_Coriolis_v(double *Dv_nAB, double Dt, sw *SW);
void do_channel(sw *SW);
void do_AdamsBashforth(double *Dvar_nAB, double *Dvar_n, double *Dvar_nM1, double *
    Dvar_nM2, int n, map *M);
void do_AdamsBashforth_variable_Dt(double *Dvar_nAB, double *Dvar_n, double *Dvar_nM1,
    double *Dvar_nM2, double Dt_n, double Dt_nM1, double Dt_nM2, int n, map *M);
double compute_Dt(double Courant, sw *SW);
void add_vortex(double x_vortex, double y_vortex, double vortex_radius, sw *SW);

// Shallow world free memory functions
void destroy_world(sw *SW);

// Postprocessing functions
void save_tr_csv(char *fname, double time, int ti, sw *SW);
void save_for_paraview(char *fname, double time, int ti, sw *SW);

// Miscellaneous functions
void create_unitary(sw *SW);
void create_Jupiter(sw *SW);
void create_Earth(sw *SW);
void add_droplets(int number_of_drops, int n, int n_period, int n_offset, int n_start,
    int n_end, sw *SW);
```

B.2 Main function

`main.c` is where the initialization happens before the core loop.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "mpi.h"

#include "sppde.h"
#include "sw.h"


int main(int argc, char **argv) {

        sw myworld;
        int nx,ny,npx,npy,N;

        checkr(MPI_Init(&argc,&argv),"init"); // Initialize MPI execution environment
            and check result
        init_pprintf(-1,1,1,"/tmp/"); // Set parallell printing options and log
            directory; second 1 means proc. 0 prints to stdout

        srand(time(NULL)); // PRAT // For droplets

        pprintf("PROGRAM STARTED\n");
        // for (int N=10;N<=90;N+=40 ) {
                cr_reset(); // Init profiling tool
                if ((argc-1)!=5) {
                        CRASH("ERR: Incorrect nuber of arguments. Pass 5 by calling '
                            mpirun sw nx ny npx npy n':\n\tnx (number of points in x)\n\
                            tny (number of points in y)\n\tnpx (number of procs in x)\n\
                            tnpy (number of procs in y)\n\tn (number of time steps)\n");
                             // QRAT
                }

                nx =atoi(argv[1]);
                ny =atoi(argv[2]);
                npx=atoi(argv[3]);
```

```
                npy=atoi(argv[4]);
                N  =atoi(argv[5]);

        create_world(nx,ny,npx,npy,&myworld); // Set up a situation

                do_core_loop(N,&myworld); // Solve the situation

        destroy_world(&myworld); // Free memory

                cr_info(); // Print profiling data
        // }

        end_pprintf(); // End print jobs
        MPI_Finalize(); // Terminate MPI environment

        return(0);
}
```

B.3 Initialization functions

With `create_world(...)` included in `sw_init`, the variables of `sw` are initialized.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"

#include "sppde.h"
#include "sppde_extensions.h"
#include "sw.h"

void create_world(int nx, int ny, int npx, int npy, sw *SW)

```

`create_world(...)` revised on 01/06/2018 by Arnau Prat Gasull.

Description

Generates a Shallow World with user data.

Notes

At the time of writing only the `DROP` and `PLANET` problem types are supported. These are explained below.

`DROP` problem type

A Shallow World of problem type `DROP` is periodic in both dimensions, which define a plane. The domain is sliced into rectangular node-centered cells. One must define the start and end coordinates as well as the range of longitudes and latitudes. Set longitude and latitude range to 0 for non varying Coriolis parameter or gravity. If the range is not 0, then these vary linearly according to latitude.

`PLANET` problem type

A Shallow World of problem type `PLANET` has spheroidal coordinates which have been created by distributing uniformly the longitudes and the planetographic latitudes. The domain is only periodic in the x direction and a channel boundary condition is achieved by calling `do_channel(...)` in the loop when the problem type is not `DROP`. The spatial coordinates are stored in the fiels of `sw` while the angles can be accessed by means of `LONS1(...)`, `LONS2(...)`, `LONC(...)`, `GLATS1(...)`, `GLATS2(...)`, `GLATC(...)`... *which are not in memory and are generated with* `get_linspace_val(...)`.

Parameters

Below follows a brief description of the variables:
`sw *SW` (input/output) The Shallow World.

Notes

The geometric information is replicated in all the processors, the fields are distributed...
The reader will note that `gl0`, `gl1`, `np4` are arrays of length 4. This is a requirement of the `sppde` library. The design of this library assumes that information is indexed from element 1, so information about in the horizontal direction (x axis) is presented at the second position of the array (because in C the index of the first element is 0, the index for the second position is 1). The third and fourth position of the array contain information about the domain in the y and z direction, respectively.
The intrinsic properties of the domain of the Shallow World are "stored" in the `m` field of the tuple. The `sppde` defines `createM(...)`, which takes the following parameters:
`int nd` (input) The number of dimensions. For a bidimensional domain set `nd` to 2.
`int *gl0` (input) The array of indexing start numbers.
`int *gl1` (input) The array of indexing end numbers.

int *np (input) The number of processors per axis.
int sh (input) The size of the halo per side of the domain. Set sh to 2 if the Superbee TVD scheme is used.
int *per (input) The domain periodicity information.

B.4 Coordinate-related functions

Coordinate information is created with the functions in sw_coords.c.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"

#include "sppde.h"
#include "sppde_extensions.h"
#include "sw.h"

void init_coords_ortho(sw *SW)

```

init_coords_ortho(...) revised on 01/06/2018 by Arnau Prat Gasull.

Description

Initializes the coordinate information of the Shallow World according to a uniform distribution of the points in the x and the y axes independently. The result is an orthogonal/rectilinear coordinate system. Matrices XS(...), XC(...), YS(...), YC(...)... are initialized.

Parameters

Below follows a brief description of the variables:
sw *SW (input/output) The Shallow World.

Notes

By calling this function, the user can initialize the distances of the points if the domain is planar. It can also be used to initialize a uniform distribution of angles which may be reused in the creation of the lengths of an ellipsoid coordinate system, as seen in init_coords_spheroid(...).

```

void init_coords_spheroid(sw *SW)

```

init_coords_spheroid(...) revised on 01/06/2018 by Arnau Prat Gasull.

Description

Initializes the coordinate information of the Shallow World according to an ellipsoidal coordinate system with uniformly distributed latitudes and longitudes.

Parameters

Below follows a brief description of the variables:
sw *SW (input/output) The Shallow World.

Notes

The precision of the lengths on the surface, stored in XS(...), XC(...), YS(...), YC(...)..., are very dependant on the precision of DXS(...), DXC(...), as they are generated by using cumsum_x(...) and cumsum_y(...). This means that for more points inbetween latitudes, the more precise is the "differential" e.g. DXS(...) and the more accurate is the cumulative sum. *Expect errors of* 10^3 *to* 10^4 *for differentials of* 10^6 *to* 10^7. This method is much more efficient than using iterative algorithms such as the Vincenty's, which cannot also be used to create domains of great ranges of longitudes or at the poles (see get_vincenty(...)).
One can take as an example init_coords_spheroid(...) in order to create a general mesher, for all geometries, as the algorithm presented is as generalized as possible, though some assumptions have been made.

Related theory

The perimeter of an ellipse cannot be analytically solved and therefore the meridian arc is difficult to compute, and even more using MPI! Therefore, one has to use approximations such as Ramanujan approximations, infinite series or Advanced numerical methods, to find the arc of an ellipse or the perimeter.
Math libraries such as Cephes or the GNU Scientific Library come with functions that compute complete and/or incomplete elliptic integrals of the second kind[1], which can be used to determine each of the processors' arc lengths. The elliptic integral

[1] See http://mathworld.wolfram.com/EllipticIntegraloftheSecondKind.html.

that needs to be solved is shown below:

$$ds = \int_0^{\varphi_C} R_P \sqrt{1 - \left(1 - \varepsilon^2\right) \sin^2 \varphi_C} d\varphi_C$$

The easiest way to compute the meridian arc is to overwrite the orthogonal coordinates with the values found using the following formula:

$$\frac{ds}{d\varphi_C} = R_P \sqrt{1 - \left(1 - \varepsilon^2\right) \sin^2 \varphi_C}$$

Once this is solved one needs to sum all the increments values up to the point whose position on the meridian is desired. This is, in fact, the relative arc length (with respect to `SW->M->l0`). In order to know the position of the point with respect to (x_0, y_0), halo updates are required. The values adjacent to the cell `SW->M->gl0` have to be summed to all of the points of the domain. Once done, a halo update might be done to update the values of the halo with its real positions.
It is, however, easy to compute the zonal distance as the cross section of an spheroid (viewed from azimuth) is a circumference. Because the perimeter (as well as the arc) of a sphere can be determined analitically by using $ds = 2\pi d\varphi$, a different treatment of matrices `XS(...)`, `XC(...)`... and `YS(...)`, `YC(...)`... might be carried out, *though it has not been done here*.
To sum up, in order to compute the coordinates on a spheroid, a uniform distribution of angles is carried out. Then, lengths on the surface are computed and stored (overwritten) on the matrices containing the uniform distribution of angles. Note that angular positions are accessible by using `GLATS1(...)`, `GLATS2(...)`, `GLATC(...)`... and `LONS1(...)`, `LONS2(...)`, `LONC(...)`... which reference `lonS(...)`, `lonC(...)`, `glatC(...)` and `glatS(...)`, which in turn make use of `get_linspace_val(...)` at their definition because *the distribution of angles is uniform*.

B.5 Solver core

The core of the solver can be found in `sw_core.c`.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"

#include "sppde.h"
#include "sppde_extensions.h"
#include "sw.h"

// MAIN LOOP

void do_core_loop(int N, sw *SW) {

```

`do_core_loop(...)` revised on 23/05/2018 by Arnau Prat Gasull.

Description

Runs the time iterations, a.k.a. performs the operations to output the solution. In fact this is the core of the program.

Parameters

Below follows a brief description of the variables:
`int N` (input) The number of time steps.
`sw *SW` (input/output) The Shallow World.

```

        int info=1;

    int i, j;

    // Auxiliary matricese
    double *Deta_nAB, *Deta_n, *Deta_nM1, *Deta_nM2;
    double *Du_nAB, *Du_adv, *Du_pres, *Du_n, *Du_nM1, *Du_nM2;
    double *Dv_nAB, *Dv_adv, *Dv_pres, *Dv_n, *Dv_nM1, *Dv_nM2;

    // Simulation parameters
    double Courant_max = 0.25;  // Maximum Courant (this is restrictive)

        int n;// Current time step and number of time steps
        double t0 = 0, t = t0;
        double local_Dt_n, Dt_n, Dt_n_CFL, Dt_nM1 = 0, Dt_nM2 = 0;

    // Output variables
```

```
    int frame_num=0;

    // Allocate auxiliary matrices
    Deta_nAB = dmem(SW->M);
    Deta_n = dmem(SW->M);
        Deta_nM1 = dmem(SW->M);
        Deta_nM2 = dmem(SW->M);
    Du_nAB = dmem(SW->M);
    Du_adv = dmem(SW->M);
    Du_pres = dmem(SW->M);
    Du_n = dmem(SW->M);
        Du_nM1 = dmem(SW->M);
        Du_nM2 = dmem(SW->M);
    Dv_nAB = dmem(SW->M);
    Dv_adv = dmem(SW->M);
    Dv_pres = dmem(SW->M);
    Dv_n = dmem(SW->M);
        Dv_nM1 = dmem(SW->M);
        Dv_nM2 = dmem(SW->M);

    pprintf("HALO UPDATE ALL FIELDS/INITIALIZE BOUNDARY CONDITIONS\n");

    halo_update(SW->u,SW->M); // Update halo for safety
    halo_update(SW->v,SW->M); // Update halo for safety
    halo_update(SW->eta,SW->M); // Update halo for safety
    halo_update(SW->hB,SW->M); // Update halo for safety
        if(SW->probdef!=DROP) do_channel(SW);

        if (info>2) {
                stats_scaf(SW->eta,SW->M,"eta"); // Print field information
                stats_scaf(SW->u,SW->M,"u"); // Print field information
                stats_scaf(SW->v,SW->M,"v"); // Print field information
        }

    pprintf("START OF TIME STEPS AT t=%5.2e\n",t0);

    for(n=1;n<=N;n++) {

        cr_start("time_step",0);

        if(info) pprintf("TIME STEP #%i\t",n);

        if(n%10==1 || n==N) { // Save every 10 time steps
            save_for_paraview("/tmp/test", t, frame_num, SW);
            save_tr_csv("/tmp/csvtest", t, frame_num, SW);
            frame_num++;
         }
```

Add a vortex.

```
                if(info>1) pprintf("probdef=%d");

                if(n%20==1 && n<2500) {
                        if(SW->probdef==DROP){
                                add_vortex(get_val(SW->xc, 15, 15, SW->M), get_val(SW->
                                    yc, 15, 15, SW->M), 1e6, SW); // Positions
                        } else {
                                add_vortex((SW->lon0+SW->lon1)/2.0, (SW->glat0+SW->glat1
                                    )/2.0, 4e6, SW); // Longitude and latitude
                        }
                }
```

Set the Dt of the current time step. The value can be user-defined or determined from the output of the last time step by taking advantage of the Courant condition.

```
        // local_Dt_n = f(U,V,mesh);
        cr_start("compute_Dt",0);
        local_Dt_n = compute_Dt(Courant_max, SW);
        MPI_Allreduce(&(local_Dt_n),&Dt_n_CFL,1,MPI_DOUBLE,MPI_MIN,MPI_COMM_WORLD); //
            Get the minimum Dt of all processes
```

```
94          Dt_n = 60;
95          if (info) pprintf("WITH Dt_n_CFL=%.6e Dt=%.6e\n",Dt_n_CFL,Dt_n);
96          cr_end("compute_Dt",0);
97
98                  if(Dt_n>=Dt_n_CFL) {
99                          pprintf("Current time step is not valid. EXITING PROGRAM\n");
100                         break;
101                 }
102
```

Compute the current Deta from the conservation of h by means of the Superbee TVD scheme.

```
103         // Deta_n = f(U,V,mesh,Dt_n);
104         cr_start("solve_cons_h",0);
105         solve_cons_h(Deta_n, Dt_n, SW);
106         cr_end("solve_cons_h",0);
107
108                 if(info>2) stats_scaf(Deta_n,SW->M,"Deta_n"); // Print field information
109
```

Compute the true value of Deta by using an Adams–Bashforth method.

```
110         // Deta_nAB = f(Deta_nAB, Deta_n, Deta_nM1, Deta_nM2, Dt_n, Dt_nM1, Dt_nM2, n )
111         cr_start("ABashforth",0);
112         do_AdamsBashforth(Deta_nAB, Deta_n, Deta_nM1, Deta_nM2, n, SW->M);
113         cr_end("ABashforth",0);
114
115                 if(SW->probdef==PLANET)
116                         forall(i,j,SW->M)
117                                 ac(Deta_nAB,i,j,SW->M)-=(Dt_n*sin(GLATC(i,j))/rZ(GLATC(i
                                        ,j))*H(i,j)*0.5*(V(i,j)+V(i,j-1)));
118
119                 if(info>2) stats_scaf(Deta_nAB,SW->M,"Deta_nAB"); // Print field
                        information
120
```

Update auxiliary matrices for use e.g. in the Adams–Bashforth method.

```
121         // Deta_n = f(Deta_nM1, Deta_nM2)
122         cr_start("copy_Dvar",0);
123                 copy_matrix(Deta_nM2,Deta_nM1,0,SW->M);
124                 copy_matrix(Deta_nM1,Deta_n,0,SW->M);
125         cr_end("copy_Dvar",0);
126
```

Update the value of eta of the Shallow World.

```
127         cr_start("newval",0);
128         forall(i, j, SW->M) {
129             ETA(i,j)=ETA(i,j)+ac(Deta_nAB,i,j,SW->M); // Add increment
130         }
131         cr_end("newval",0);
132
```

Update the halos of the processors.

```
133         cr_start("halo_update",0);
134         halo_update(SW->eta,SW->M); // Update the halos
135         cr_end("halo_update",0);
136
```

Compute the current Du and Dv attributed to advection by using the Superbee TVD scheme.

```
137         cr_start("solve_adv",0);
138         solve_adv_u(Du_adv, Dt_n, SW); // Compute Du_adv from the advection in the x
                axis by means of the Superbee TVD scheme
139         solve_adv_v(Dv_adv, Dt_n, SW); // Compute Dv_adv from the advection in the y
                axis by means of the Superbee TVD scheme
140         cr_end("solve_adv",0);
141
```

Compute the current Du and Dv attributed to the distribution of pressure.

```
142         cr_start("solve_press",0);
143         solve_pres_u(Du_pres, Dt_n, SW); // Compute Du_pres from the gradient of
                pressure in the x axis
```

```
144         solve_pres_v(Dv_pres, Dt_n, SW); // Compute Dv_pres from the gradient of
                pressure in the y axis
145         cr_end("solve_press",0);
146
```

Sum the increment of Du and Dv attributed to the advection and the pressure gradients.

```
147             cr_start("newvel",0);
148         forall(i, j, SW->M) {
149             ac(Du_n,i,j,SW->M)=ac(Du_adv,i,j,SW->M)+ac(Du_pres,i,j,SW->M);// Add the
                    contributions of advection and pressure
150             ac(Dv_n,i,j,SW->M)=ac(Dv_adv,i,j,SW->M)+ac(Dv_pres,i,j,SW->M);// Add the
                    contributions of advection and pressure
151         }
152             cr_end("newvel",0);
153
154             if(info>2) {
155                     stats_scaf(Du_adv,SW->M,"Du_adv"); // Print field information
156                     stats_scaf(Dv_adv,SW->M,"Dv_adv"); // Print field information
157                     stats_scaf(Du_pres,SW->M,"Du_pres"); // Print field information
158                     stats_scaf(Dv_pres,SW->M,"Dv_pres"); // Print field information
159                     stats_scaf(Du_n,SW->M,"Du_n"); // Print field information
160                     stats_scaf(Dv_n,SW->M,"Dv_n"); // Print field information
161             }
162
```

Compute the true values of Du and Dv by using an Adams–Bashforth method.

```
163         cr_start("ABashforth",0);
164         do_AdamsBashforth(Du_nAB, Du_n, Du_nM1, Du_nM2, n, SW->M); // Compute the
                increment to be added Du_nAB to the current u
165         do_AdamsBashforth(Dv_nAB, Dv_n, Dv_nM1, Dv_nM2, n, SW->M);// Compute the
                increment to be added Dv_nAB to the current v
166         cr_end("ABashforth",0);
167
```

Update auxiliary matrices for use e.g. in the Adams–Bashforth method.

```
168         cr_start("copy_Dvar",0);
169             copy_matrix(Du_nM2,Du_nM1,0,SW->M);
170             copy_matrix(Du_nM1,Du_n,0,SW->M);
171             copy_matrix(Dv_nM2,Dv_nM1,0,SW->M);
172             copy_matrix(Dv_nM1,Dv_n,0,SW->M);
173             cr_end("copy_Dvar",0);
174
```

Apply the Coriolis effect to the current increments Du and Dv.

```
175             /*
176             The following is what the program does when f=0.
177
178             forall(i, j, SW->M) {
179                 U(i,j)=U(i,j)+ac(Du_nAB,i,j,SW->M); // Add incrrement
180                 V(i,j)=V(i,j)+ac(Dv_nAB,i,j,SW->M); // Add incrrement
181             }
182             */
183         cr_start("newval",0);
184         solve_Coriolis_u(Du_nAB, Dt_n, SW);
185         solve_Coriolis_v(Dv_nAB, Dt_n, SW);
186         cr_end("newval",0);
187
```

Update the halos of the processors.

```
188         cr_start("halo_update",0);
189         halo_update(SW->u,SW->M); // Update the halos
190         halo_update(SW->v,SW->M); // Update the halos
191         cr_end("halo_update",0);
192
```

Update the simulation time.

```
193         t += Dt_n; // Update time
194
195         Dt_nM2 = Dt_nM1;
196         Dt_nM1 = Dt_n;
```

Apply the channel boundary condition (this could be moved up to the start of the loop).

```
                if(SW->probdef!=DROP) do_channel(SW);
```

Add droplet (changes of eta to single points) to simulate rain (this could be moved up to the start of the loop).

```
        // add_droplets(SW,1,n,5,0,0,300);

        if (info>1) {
                        stats_scaf(SW->eta,SW->M,"eta"); // Print field information
                        stats_scaf(SW->u,SW->M,"u"); // Print field information
                        stats_scaf(SW->v,SW->M,"v"); // Print field information
                }
        if (info>3) {
                        print_scaf(SW->xs, "xs", 0, SW->M,"%8.2e ");
            print_scaf(SW->xc, "xc", 0, SW->M,"%8.2e ");
                        print_scaf(SW->xv, "xv", 0, SW->M,"%8.2e ");
            print_scaf(SW->ys, "ys", 0, SW->M,"%8.2e ");
            print_scaf(SW->yc, "yc", 0, SW->M,"%8.2e ");
                        print_scaf(SW->yv, "yv", 0, SW->M,"%8.2e ");
                }
                if(info>4) {
                        print_scaf(SW->Dxs, "Dxs", 0, SW->M,"%8.2e ");
                        print_scaf(SW->Dxc, "Dxc", 0, SW->M,"%8.2e ");
                        print_scaf(SW->Dxv, "Dxv", 0, SW->M,"%8.2e ");
                        print_scaf(SW->Dys, "Dys", 0, SW->M,"%8.2e ");
                        print_scaf(SW->Dyc, "Dyc", 0, SW->M,"%8.2e ");
                        print_scaf(SW->Dyv, "Dyv", 0, SW->M,"%8.2e ");
                }
                if (info>5) {
                        print_scaf(Deta_n, "Deta_n", 0, SW->M,"%8.2e ");
                        stats_scaf(SW->u,SW->M,"u"); // Print field information
                print_scaf(Du_adv, "Du_adv", 0, SW->M,"%8.2e ");
                print_scaf(Du_pres, "Du_pres", 0, SW->M,"%8.2e ");
                print_scaf(Du_n, "Du_n", 0, SW->M,"%8.2e ");
                stats_scaf(SW->v,SW->M,"v"); // Print field information
                print_scaf(Dv_adv, "Dv_adv", 0, SW->M,"%8.2e ");
                print_scaf(Dv_pres, "Dv_pres", 0, SW->M,"%8.2e ");
                print_scaf(Dv_n, "Dv_n", 0, SW->M,"%8.2e ");
        }
                if(info>1) pprintf("\n");
        cr_end("time_step",0);

    }

    pprintf("END OF TIME STEPS AT t=%5.2e\n",t);

    if (info>0) {
        stats_scaf(SW->eta,SW->M,"eta"); // Print field information
        stats_scaf(SW->u,SW->M,"u"); // Print field information
        stats_scaf(SW->v,SW->M,"v"); // Print field information
    }

    // Free memory
    free(Deta_nAB);
    free(Deta_n);
    free(Deta_nM1);
    free(Deta_nM2);
    free(Du_nAB);
    free(Du_adv);
    free(Du_pres);
    free(Du_n);
    free(Du_nM1);
    free(Du_nM2);
    free(Dv_nAB);
    free(Dv_adv);
    free(Dv_pres);
    free(Dv_n);
    free(Dv_nM1);
```

```
263     free(Dv_nM2);
264
265 } // PRAT
```

B.6 Solver core functions

The functions called in the core of the solver are available in `sw_calc.c`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include "mpi.h"
5
6  #include "sppde.h"
7  #include "sppde_extensions.h"
8  #include "sw.h"
9
10 // ACCESS MACROS FOR AUXILIARY MATRICES USED IN THE SOLVER
11
12 // Du at nM2, nM1 and n, and the contributions of the pressure and the advection at the
       current time step n and the output of the Adams--Bashforth
13 #define DU_nAB(i,j)  ac(Du_nAB,i,j,SW->M) // PRAT
14 #define DU_pres(i,j) ac(Du_pres,i,j,SW->M) // PRAT
15 #define DU_adv(i,j)  ac(Du_adv,i,j,SW->M) // PRAT
16
17 // Dv at nM2, nM1 and n, and the contributions of the pressure and the advection at the
       current time step n and the output of the Adams--Bashforth
18 #define DV_nAB(i,j)  ac(Dv_nAB,i,j,SW->M) // PRAT
19 #define DV_pres(i,j) ac(Dv_pres,i,j,SW->M) // PRAT
20 #define DV_adv(i,j)  ac(Dv_adv,i,j,SW->M) // PRAT
21
22 // Superbee TVD scheme limiter
23 #define funPsi(r) (MAX(MAX(0,MIN(2*r,1)),MAX(0,MIN(r,2)))) // QRAT
24
25 // TVD scheme related macros
26 #define B2BP(B1,B2,rP,CP) (B1 + funPsi(rP) * 0.5 * ( 1 - CP ) * ( B2 - B1 )) // QRAT
27 #define B2BM(B1,B2,rM,CM) (B2 - funPsi(rM) * 0.5 * ( 1 + CM ) * ( B2 - B1 )) // QRAT
28 #define r2rP(Bprev,Bcurr,Bpost) (((Bpost - Bcurr) != 0) ? ((Bcurr - Bprev) / (Bpost -
       Bcurr)) : 0) // QRAT
29 #define r2rM(Bcurr,Bpost,Bnext) (((Bpost - Bcurr) != 0) ? ((Bnext - Bpost) / (Bpost -
       Bcurr)) : 0) // QRAT
30
31 void solve_cons_h(double *Deta_n, double Dt, sw *SW)
32
```

`solve_cons_h(...)` revised on 01/06/2018 by Arnau Prat Gasull.

Description

Computes the increment of the surface perturbation `Deta_n` according to the equation of the conservation of h and also to the Superbee TVD scheme.

Parameters

Below follows a brief description of the variables:
`double *Deta_n` (input/output) The increment of the surface perturbation.
`double Dt` (input) The current increment of time.
`sw *SW` (input) The Shallow World.

```
33
34 void solve_adv_u(double *Du_adv, double Dt, sw *SW)
35
```

`solve_adv_u(...)` revised on 01/06/2018 by Arnau Prat Gasull.

Description

Computes the increment of the horizontal velocities `Du_adv` due to advection.

Parameters

Below follows a brief description of the variables:
`double *Du_adv` (input/output) The increment of the horizontal velocities due to advection.
`double Dt` (input) The current increment of time.

sw *SW (input) The Shallow World.

```
void solve_adv_v(double *Dv_adv, double Dt, sw *SW)
```

solve_adv_v(...) revised on 01/06/2018 by Arnau Prat Gasull.

Description
Computes the increment of the vertical velocities Dv_adv due to advection.

Parameters
Below follows a brief description of the variables:
double *Dv_adv (input/output) The increment of the vertical velocities due to advection.
double Dt (input) The current increment of time.
sw *SW (input) The Shallow World.

```
void solve_pres_u(double *Du_pres, double Dt, sw *SW)
```

solve_pres_u(...) revised on 01/06/2018 by Arnau Prat Gasull.

Description
Computes the increment of the horizontal velocities Du_pres due to the distribution of pressure in the x axis.

Parameters
Below follows a brief description of the variables:
double *Du_pres (input/output) The increment of the horizontal velocities due to the distribution of pressure.
double Dt (input) The current increment of time.
sw *SW (input) The Shallow World.

```
void solve_pres_v(double *Dv_pres, double Dt, sw *SW)
```

solve_pres_v(...) revised on 01/06/2018 by Arnau Prat Gasull.

Description
Computes the increment of the vertical velocities Du_pres due to the distribution of pressure in the y axis.

Parameters
Below follows a brief description of the variables:
double *Du_pres (input/output) The increment of the vertical velocities due to the distribution of pressure.
double Dt (input) The current increment of time.
sw *SW (input) The Shallow World.

```
void solve_Coriolis_u(double *Du_nAB, double Dt, sw *SW)
```

solve_Coriolis_u(...) revised on 01/06/2018 by Arnau Prat Gasull.

Description
Computes the contribution to the horizontal velocities of the Coriolis effect..

Parameters
Below follows a brief description of the variables:
double *Du_nAB (input) The increment of the horizontal velocities after calling do_AdamsBashforth(...).
double Dt (input) The current increment of time.
sw *SW (input/output) The Shallow World.

Notes
The term arising from expressing the conservation of momentum in ellipsoidal coordinates is integrated here (only what affects the horizontal velocities).

```
void solve_Coriolis_v(double *Dv_nAB, double Dt, sw *SW)
```

solve_Coriolis_v(...) revised on 01/06/2018 by Arnau Prat Gasull.

Description
Computes the contribution to the vertical velocities of the Coriolis effect..

Parameters
Below follows a brief description of the variables:
double *Dv_nAB (input) The increment of the vertical velocities after calling do_AdamsBashforth(...).
double Dt (input) The current increment of time.
sw *SW (input/output) The Shallow World.

Notes
The term arising from expressing the conservation of momentum in ellipsoidal coordinates is integrated here (only what affects the vertical velocities).

```
51
52 void do_channel(sw *SW)
53
```

do_channel(...) revised on 01/06/2018 by Arnau Prat Gasull.

Description
Sets the boundary conditions for a wall at the top and at the bottom of the domain.

Parameters
Below follows a brief description of the variables:
sw *SW (input/output) The Shallow World.

Improvements
Create a function add_wall(SW, a, b) which, when called with a=1, a wall along the x is created. For a=2, a wall is placed along the y axis. b=0 places a wall at the left boundary and b=1 at the right. This syntax follows the same nomenclature as the rest of the program. Then define a macro or a function that replaces add_walls(SW) with calls to add_wall(SW,1,0), add_wall(SW,1,1), add_wall(SW,2,0) and add_wall(SW,2,1).

```
54
55 void do_AdamsBashforth(double *Dvar_nAB, double *Dvar_n, double *Dvar_nM1, double *
       Dvar_nM2, int n, map *M)
56
```

do_AdamsBashforth(...) revised on 01/06/2018 by Arnau Prat Gasull.

Description
Computes Dvar_nAB, which is the increment to be added to the current variable according to Adams–Bashforth expressions. These require the two past results of Dvar_n, which are identified as Dvar_nM1 and Dvar_nM2.

Parameters
Below follows a brief description of the variables:
double *Dvar_nAB (output) The real increment of the variable.
double *Dvar_n (input) The current increment of the variable.
double *Dvar_nM1 (input) The past increment of the variable.
double *Dvar_nM2 (input) The increment of the variable used two time steps before.
sw *SW (input) The Shallow World.

Notes
Do NOT use this function for variable Δt. Instead, use do_AdamsBashforth_variable_Dt(...) for variable Δt. This function also requires n in order to apply the Euler method, the two-step Adams–Bashforth or the three-step Adams–Bashforth accordingly.

```
57
58 void do_AdamsBashforth_variable_Dt(double *Dvar_nAB, double *Dvar_n, double *Dvar_nM1,
       double *Dvar_nM2, double Dt_n, double Dt_nM1, double Dt_nM2, int n, map *M)
59
```

do_AdamsBashforth_variable_Dt(...) revised on 08/06/2018 by Arnau Prat Gasull.

Description
Computes Dvar_nAB, which is the increment to be added to the current variable according to Adams–Bashforth expressions. These require the two past results of Dvar_n, which are identified as Dvar_nM1 and Dvar_nM2 as well as the two past increments of time Dt_nM1 and Dt_nM2. The function also requires n in order to apply the Euler method, the two-step Adams–Bashforth or the soon-to-be-implemented three-step Adams–Bashforth accordingly.

Parameters
Below follows a brief description of the variables:
double *Dvar_nAB (output) The real increment of the variable.
double *Dvar_n (input) The current increment of the variable.
double *Dvar_nM1 (input) The past increment of the variable.
double *Dvar_nM2 (input) The increment of the variable used two time steps before.
double Dt_n (input) The current increment of time.

double Dt_nM1 (input) The last increment of time.
double Dt_nM2 (input) The increment of time used two time steps before.
int n (input) The current timestep.
sw *SW (input) The Shallow World.

```
double compute_Dt(double Courant, sw *SW)
```

`compute_Dt(...)` revised on 08/06/2018 by Arnau Prat Gasull.

Description
Returns the required Δt to be added to the current time t to perform the simulation. From the definition of the Courant number in two dimensions, the function returns the smallest Dt of the domain. For velocity fields of 0, the functions computes Dt according to the maximum possible velocity, given by $u_{\max}$, $v_{\max} = \sqrt{gD}$.

Parameters
Below follows a brief description of the variables:
double Courant (input) The maximum allowed Courant number. Smaller courant numbers are more restrictive.
sw *SW (input) The Shallow World.

```
void add_vortex(double x_vortex, double y_vortex, double vortex_radius, sw *SW)
```

`add_vortex(...)` revised on 07/06/2018 by Arnau Prat Gasull.

Description
Adds a perturbation of radius `vortex_radius` on the surface of the domain at position (`x_vortex`,`y_vortex`).

Parameters
Below follows a brief description of the variables:
double x_vortex (input) The starting longitude.
double y_vortex (input) The starting latitude.
double vortex_radius (input) The ending longitude.
sw *SW (input) The Shallow World.

Notes
When computing the distance between the centre of the perturbation to a point of a region of a spheroid, a formula meant for spheres is used (formula number 80 of Williamson 1992–A Standard Test Set for Numerical Approximations to the Shallow Water Equations in Spherical Geometry):

B.7 Termination functions

Memory deallocation is done with `destroy_world(...)`, found in `sw_end`.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"

#include "sppde.h"
#include "sw.h"

void destroy_world(sw *SW)
```

`destroy_world(...)` revised on 09/06/2018 by Arnau Prat Gasull.

Description
Deallocs the memory.

Parameters
Below follows a brief description of the variables:
sw *SW (input/output) The Shallow World.

B.8 Related functions

For-general use functions are defined in `sw_tools.c(...)`.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"

#include "sppde.h"
#include "sppde_tinyexpr.h"
#include "sppde_parser.h"
#include "sw.h"

// void vel2coords(double u, double v, double *lon, double *glat, double Dt, sw *SW){
//     (*glat)+=((Dt*v)/rM(*glat));
// }

double linterp_core(double x_i, double *x, double *y, int *i0, int *i1)

```

`linterp_core(...)` revised on 01/06/2018 by Arnau Prat Gasull.

Description

Returns the linearly interpolated value of the function $y = y(x)$ at point x_i.

Parameters

Below follows a brief description of the variables:
`double x_i` (input) The value at which the interpolation is done (function evaluation point).
`double *x` (input) The points whose image of $y = y(x)$ is known.
`double *y` (input) The images of x.
`int *i0` (input/output) The search start index of `x`.
`int *i1` (input/output) The search end index of `x`.

Notes

This function only does an out-of-bonds check. The interpolated value is found using only one while loop.
The search start index is updated with the index at which the interpolated value has been found.

```c

double linterp(double x_i, double *x, double *y, int n)

```

`linterp(...)` revised on 01/06/2018 by Arnau Prat Gasull.

Description

Returns the linearly interpolated value of the function $y = y(x)$ at point x_i.

Parameters

Below follows a brief description of the variables:
`double x_i` (input) The value at which the interpolation is done (function evaluation point).
`double *x` (input) The points whose image of $y = y(x)$ is known.
`double *y` (input) The images of x.
`int n` (input) The length of vectors `x` and `y`.

Notes

This is a redefinition of `linterp_core(...)` for easy use (starting and ending indeces are 0 to length of the array).

```c

void linterp_vec(double *x_i, double *y_i, int n_i, double *x, double *y, int n)

```

`linterp_vec(...)` revised on 01/06/2018 by Arnau Prat Gasull.

Description

Returns y_i, which are the linearly interpolated values of the function $y = y(x)$ at points x_i.

Parameters

Below follows a brief description of the variables:
`double *x_i` (input) The values at which the interpolation is done (function evaluation points).
`double *y_i` (output) The images of $y = y(x)$ at points x_i.
`int n_i` (input) The lenght of vectors `x_i` and `y_i`.
`double *x` (input) The points whose image of $y = y(x)$ is known.
`double *y` (input) The images of x.
`int n` (input) The length of vectors `x` and `y`.

Notes
This function assumes that the values are sorted by increasing order. This calls `linterp_core(...)` as this function returns updated indices (the updated index is that of the last found value of `y`).

```
23
24 void read_zw(char *fname, sw *SW)
25
```

`read_zw(...)` revised on 01/06/2018 by Arnau Prat Gasull.

Description
Reads a zonal wind file and updates the array of the Shallow World dedicated to it.

Parameters
Below follows a brief description of the variables:
`char *fname` (input) The name of the file.
`sw *SW` (input/output) The Shallow World.

Special requirements
Requires `sppde_tinyexpr` and `sppde_parser`.

```
26
27 void copy_matrix(double *matrixCopy, double *matrixInput, int copy_halo, map *M);
28
```

`copy_matrix(...)` revised on 28/05/2018 by Arnau Prat Gasull.

Description
Copies the matrix `matrixInput` to `matrixCopy`, both characterized by `M`. The contents of the halo are passed to `matrixCopy` if `copy_halo=1`.

Parameters
Below follows a brief description of the variables:
`double *matrixCopy` (output) The copied matrix with the values of `matrixInput`.
`double *matrixInput` (input) The origin of the values that are passed to `matrixCopy`.
`double *copy_halo` (input) If `copy_halo=0`, only the domain is copied. If `copy_halo=1`, the values of the halo are also copied.

Use case
This function can be used for passing values between time steps, as the function copies the current value of a given variable `matrixInput` to `matrixCopy`, which is the value of the previous time step.

```
29
30 double get_vincenty(double lon0, double lat0, double lon1, double lat1, double a, double
       b)
31
```

`get_vincenty(...)` revised on 28/05/2018 by Arnau Prat Gasull.

Description
Computes the geodetic between points (λ_0, φ_0) and (λ_1, φ_1) according to the Vincenty formulae [12].

Parameters
Below follows a brief description of the variables:
`double lon0` (input) The starting longitude.
`double lat0` (input) The starting latitude.
`double lon1` (input) The ending longitude.
`double lat1` (input) The ending latitude.
`double a` (input) The semi-major axis of the ellipse.
`double b` (input) The semi-minor axis of the ellipse.

Use case
Instead of using the haversine formula on an ellipsoid to determine the geodetic, one shall use the Vincenty algorithm to determine with much more precision the geodetic between two points on a ellipsoid given the angles λ and ϕ. The geodetic is used in Shallow Worlds to compute e.g. the area of influence of a vortex.

Notes
The Vincenty algorithm may fail to converge for points close to the poles and, while it converges very fast for points close to the equator, the convergence rate decreases for increasing latitudes. Also, do not use this formula to compute the distance between antipodal points as it will produce nonsense results. Only use `get_vincenty(...)` when the distance between relatively close points (such as in the case of boundary to center points of a vortex) is needed. The GeographicLib implements an algorithm that computes the geodetic even when the Vincenty algorithm fails. It may not be as fast as `get_vincenty(...)`.

```
double get_arc_ellipse(double a1, double rP, double epsilon2)
```

`get_arc_ellipse(...)` revised on 28/05/2018 by Arnau Prat Gasull.

Description
Returns the arc of ellipse from 0 to a1, in radians.

Parameters
Below follows a brief description of the variables:
`double a1` (input) The end angle.
`double rP` (input) The semi-minor axis (or b) of the ellipse (polar radius).
`double epsilon2` (input) For an ellipse of radius a and b, $\varepsilon = a/b$. ε is the quotient between the equatorial radius `rE` and the polar radius `rP`.

```
double get_arc_circumf(double a1, double r)
```

`get_arc_circumf(...)` revised on 28/05/2018 by Arnau Prat Gasull.

Description
Returns the arc of circumference from 0 to a1, in radians.

Parameters
Below follows a brief description of the variables:
`double a1` (input) The end angle.
`double r` (input) The radius of the circumference.

```
double get_linspace_val(int i,double x0,double x1,int nx)
```

`get_linspace_val(...)` revised on 23/05/2018 by Arnau Prat Gasull.

Description
Returns the value at index i of a vector of n_x values uniformly distributed from x_0 to x_1. For $i = 0$ the value returned is $x0$.

Parameters
Below follows a brief description of the variables:
`int i` (input) The element of the vector to be returned.
`double x0` (input) The first value of the vector, found at index $i = 0$.
`double x1` (input) The last value of the vector, found at index $i = n_x$.
`int nx` (input) The number of values in the vector

Use cases
At the core of `init_coords_ortho(...)`, this function is used to distribute the coordinates uniformly. This function is used to create the coorinates of the points of the halo to avoid calls to `halo_update(...)`, so indices $i < 0$ and $i > n_x$ are also valid. This function is also used in the definition `lonS`, `lonC`, `lonV`, `glatS`, `glatC` and `glatV`.

```
void generate_scaf(double *scaf, double (*funcio)(double,double,double), int stgx, int
    stgy, double t, sw *SW)
```

`generate_scaf(...)` revised on 23/05/2018 by Arnau Prat Gasull.

Description
Generates a scalar field according to a mathematical function defined outside of this function.

Parameters
Below follows a brief description of the variables:
`int scaf` (output) The element of the vector to be returned.
`double (*funcio)(double,double,double)` (input) The name of the function $f(x, y, t)$ to call which will be used to compute the values of `scaf`.
`int stgx` (input) If 0, centered coordinates will be used. If 1, staggered coordinates will be used.
`int stgx` (input) If 0, centered coordinates will be used. If 1, staggered coordinates will be used.
`double t` (input) The current simulation time.
`sw *SW` (input) The Shallow World.

```
double generate_val(int i, int j, double (*funcio)(double,double,double), int stgx, int
    stgy, double t, sw *SW)
```

```
46
```

`generate_val(...)` revised on 23/05/2018 by Arnau Prat Gasull.

Description

Generates a scalar field according to a mathematical function defined outside of this function.

Parameters

Below follows a brief description of the variables:
`int i` (input) The position of the element in the x direction.
`int j` (input) The position of the element in the y direction.
`double (*funcio)(double,double,double)` (input) The name of the function $f(x, y, t)$ to call which will be used to compute the values of `scaf`.
`int stgx` (input) If 0, centered coordinates will be used. If 1, staggered coordinates will be used.
`int stgx` (input) If 0, centered coordinates will be used. If 1, staggered coordinates will be used.
`double t` (input) The current simulation time.
`sw *SW` (input) The Shallow World.

B.9 Other functions

Miscellaneous functions are written in `sw_misc.c`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include "mpi.h"
5
6  #include "sppde.h"
7  #include "sppde_extensions.h"
8  #include "sw.h"
9
10 void create_unitary(sw *SW)
11
```

`create_unitary(...)` revised on 01/06/2018 by Arnau Prat Gasull.

Description

Initializes the Shallow World parameters according to a planet of radius, angular speed and gravity pull of 1.

Parameters

Below follows a brief description of the variables:
`sw *SW` (input/output) The Shallow World.

```
12
13 void create_Jupiter(sw *SW)
14
```

`create_Jupiter(...)` revised on 01/06/2018 by Arnau Prat Gasull.

Description

Initializes the Shallow World parameters according to Jupiter's.

Parameters

Below follows a brief description of the variables:
`sw *SW` (input/output) The Shallow World.

```
15
16 void create_Earth(sw *SW)
17
```

`create_Earth(...)` revised on 01/06/2018 by Arnau Prat Gasull.

Description

Initializes the Shallow World parameters according to Earth's.

Parameters

Below follows a brief description of the variables:
`sw *SW` (input/output) The Shallow World.

```
18
19 void add_droplets(int number_of_drops, int n, int n_period, int n_offset, int n_start,
       int n_end, sw *SW)
```

```
20
```

`add_droplets(...)` revised on 23/05/2018 by Arnau Prat Gasull.

Description
Returns a modified η by adding a perturbation of random intensity at random locations inside the domain.

Parameters
Below follows a brief description of the variables:
`int number_of_drops` (input) The number of droplets at each function call. It can be a random integer.
`int n` (input) The current simulation time step.
`int n_period` (input) The number of time steps between drops.
`int n_offset` (input) The adjustment factor.
`int n_start` (input) The starting time step.
`int n_end` (input) The ending time step.
`sw *SW` (input/output) The Shallow World.

Use cases
Though it is not meant to be used on planetary athmospheres, this function has been used for visualization tests and to verify if the results match the reality.

B.10 Post-processing functions

In `sw_postproc.c` the user will find the functions that output the files for post-processing.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"

#include "sppde.h"
#include "sppde_extensions.h"
#include "sw.h"

#define SCAF(i,j) ac(scafi,j,MJ_) // distributed generic field
#define GSCAF(i,j) ac(gscaf,i,j,MJ_) // local (gathered) field

void save_for_paraview(char *fname, double time, int ti, sw *SW)

```

`save_for_paraview(...)` revised on 09/06/2018 by Arnau Prat Gasull.

Description
Returns a transposed file that is understood by SWreader.

Parameters
Below follows a brief description of the variables:
`char *fname` (input) The name of the out file.
`double time` (input) The current simulation time.
`int ti` (input) The timestep.
`sw *SW` (input/output) The Shallow World.

```c


void save_tr_csv(char *fname, double time, int ti, sw *SW)

```

`save_tr_csv(...)` revised on 09/06/2018 by Arnau Prat Gasull.

Description
Returns a transposed csv file.

Parameters
Below follows a brief description of the variables:
`char *fname` (input) The name of the out file.
`double time` (input) The current simulation time.
`int ti` (input) The timestep.
`sw *SW` (input/output) The Shallow World.

Use cases

The user has to use csvtool to transpose the data.

B.11 The author's contribution to sppde

The functions that may be incorporated to sppde are found in sppde_extensions.c.

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include "sppde.h"
#include "sppde_extensions.h"

```

The code in sppde_extensions.h and sppde_extensions.c is a compilation of tools that complement the current sppde library. This functions may be integrated into the library in the future, after they have been validated.

```


```

The functions below are an improvement over Manel Soria's halo update to support full-duplex networks. The old halo update is called legacy_halo_update(...).

```
void easy_sr(int nb,double *bs,double *br,int ndata)

```

easy_sr(...) revised on 08/06/2018 by Arnau Prat Gasull.

Description

Replaces the old easy_s(...) and easy_r(...) so processors can exchange information in a full-duplex network. This means that the processors can be senders and receivers at the same time.

Parameters

Below follows a brief description of the variables:
double nb (input) The neighbour to exchange information.
double *bs (input/output) The sending [previously allocatted] buffer.
double *br (input/output) The receiving [previously allocatted] buffer.
int ndata (input) The number of elements to be sent or received. The number of elements that are exchanged are always the same for the sending and receiving buffers.

Notes

This function makes use of MPI_Sendrecv(...).

```

void halo_update_y(double *x,map *M)

```

halo_update_y(...) revised on 08/06/2018 by Arnau Prat Gasull.

Description

Updates the processor's north and south halos when the number of processors in the column is >1.

Parameters

Below follows a brief description of the variables:
double *x (input/output) The matrix/field to be updated.
map *M (input) The scaf's initrinsic properties.

```

void halo_update_x(double *x,map *M)

```

halo_update_x(...) revised on 08/06/2018 by Arnau Prat Gasull.

Description

Updates the processor's east and west halos when the number of processors in the row is >1.

Parameters

Below follows a brief description of the variables:
double *x (input/output) The matrix/field to be updated.
map *M (input) The scaf's initrinsic properties.

```

void halo_update_single_map_y(double *x,map *M)
```

`halo_update_single_map_y(...)` revised on 08/06/2018 by Arnau Prat Gasull.

Description
Updates the processor's north and south halos if in the column there is only one processor.

Parameters
Below follows a brief description of the variables:
`double *x` (input/output) The matrix/field to be updated.
`map *M` (input) The `scaf`'s initrinsic properties.

```
void halo_update_single_map_x(double *x,map *M)
```

`halo_update_single_map_x(...)` revised on 08/06/2018 by Arnau Prat Gasull.

Description
Updates the processor's east and west halos if in the row there is only one processor.

Parameters
Below follows a brief description of the variables:
`double *x` (input/output) The matrix/field to be updated.
`map *M` (input) The `scaf`'s initrinsic properties.

```
void halo_update(double *x,map *M)
```

`halo_update(...)` revised on 08/06/2018 by Arnau Prat Gasull.

Description
Updates the processor's halos.

Parameters
Below follows a brief description of the variables:
`double *x` (input/output) The matrix/field to be updated.
`map *M` (input) The `scaf`'s initrinsic properties.

Notes
This function distinguishes between `np=1` and `np>1`. It could, however, also discriminate cases where `npx=1` or `npy=1` because currently, periodic boundary conditions do not work for when in the given axis there is only 1 processor.

The function below may be integrated with `ac(...)` for improving the use.

```
double get_val(double *scaf, int i, int j, map *M)
```

`get_val(...)` revised on 08/06/2018 by Arnau Prat Gasull.

Description
Returns the value of (i, j) of a `scaf` stored in memory and defined by `M`, even if the point is not owned by the processor.

Parameters
Below follows a brief description of the variables:
`double *scaf` (input) The scalar field stored in memory of which the value is desired.
`int i` (input) The row index of the matrix `scaf`.
`int j` (input) The column index of the matrix `scaf`.
`map *M` (input) The `scaf`'s initrinsic properties.

Notes
This function makes two network communications. The first one, `MPI_Allreduce(...)` is needed to know which processor owns the point and therefore, to know the processor that will broadcast the value to the other processors through `MPI_Bcast(...)`. *If this function is integrated into `halo_update(...)`, halo updating and process comunication will not be seen by the user*.

The functions below conform a tool to compute the cumulative sum at each position of a field.

```
void cumsum_y(double *inscaf, double *outscaf, int halo_method, map *M)
```

```
33
```

`cumsum_y(...)` revised on 21/05/2018 by Arnau Prat Gasull.

Description

Computes the global cumulative sum of `inscaf` in the *y* direction and returns them in `outscaf`. `inscaf` and `outscaf` share the same map (intrinsic properties). The halo is modified according to `halo_method`.

Parameters

`inscaf` and `outscaf` may be the same, but this practice is not recommended. Below follows a brief description of the variables:
`double *inscaf` (input) The scalar field of differences.
`double *outscaf` (output) The scalar field which will be updated with the cumulative sum.
`int halo_method` (input) If 0, the cumulative sum is done excluding the halos. If 1, they are included. If 2 the cumulative sum is done on the domain points but in a fashion that gives the same results as if `halo_update_y(...)` had been called.
`map *M` (input) The `scaf`'s initrinsic properties.

Notes

For `halo_method==2`:The idea behind the code is to perform a cumulative sum on the domain and update the values of the halo as well without performing a `halo_update(...)` call, as it is an expensive operation. The values of `outscaf` are copied to `inscaf` and a cumulative sum is performed in the y direction on the points of the domain (and to the east and west sides of the halo, as they do not contribute to the sum). A call to `local2global_cumsum_y(...)` is done once the relative sum is done in each of the processors. The south and north sides of the halo are updated with the correct values after the function call. Note that the cumulative sum for the south side halo is done by summing the values "downwards" (the values have to be subtracted) instead of "upwards".

```
34
35 void cumsum_x(double *inscaf, double *outscaf, int halo_method, map *M)
36
```

`cumsum_x(...)` revised on 23/05/2018 by Arnau Prat Gasull.

Description

Performs the cumulative sum in the *x* direction, analogously to `cumsum_y(...)`.

Parameters

`inscaf` and `outscaf` may be the same, but this practice is not recommended. Below follows a brief description of the variables:
`double *inscaf` (input) The scalar field of differences.
`double *outscaf` (output) The scalar field which will be updated with the cumulative sum.
`int halo_method` (input) If 0, the cumulative sum is done excluding the halos. If 1, they are included. If 2 the cumulative sum is done on the domain points but in a fashion that gives the same results as if `halo_update(...)` had been called.
`map *M` (input) The `scaf`'s initrinsic properties.

```
37
38
39 void local2global_cumsum_y(double *scaf, int halo_method, map *M)
40
```

`local2global_cumsum_y(...)` revised on 23/05/2018 by Arnau Prat Gasull.

Description

Computes the global cumulative sum from relative cumulative `scaf`s inthe *y* direction. It is used at the core of `cumsum_y(...)`.

Parameters

Below follows a brief description of the variables:
`double *scaf` (input/output) The scalar field which will be updated with the cumulative sum.
`int halo_method` (input) If 0, the cumulative sum is done excluding the halos. If 1, they are included. If 2 the cumulative sum is done on the domain points but in a fashion that gives the same results as if `halo_update(...)` had been called.
`map *M` (input) The `scaf`'s initrinsic properties.

Use case

Suppose every processor has a relative coordinate, a.k.a. at position `M->l0[1]` the coordinate is 0 and the coordinates of the processor field are computed from this point. Assume that for dimension *x* (`[1]`) the coordinates are stored in `scaf`. After calling this function with `scaf`, `scaf` will no longer contain the coordinates referenced to its `M->l0[1]` but to `M->gl0[1]`, which means that the cumulative sum is no longer local, but global.

Implementation

Each processor *i* receives the sum of the 1 to *i* processors' relevant top rows. The difference of the value between the relevant top row and the received sum is added to all values of the processor's domain and also to the halo, if selected. Note that for the cumulative sum in the *y* direction, the top row is the one to be selected because it contains the coordinates of the points furthest from `M->l0[1]`. For `halo_method==0` and `halo_method==2`, the relevant top row is the top row of the processor's domain.

For halo_method==1, the relevant top row is that of the processor including the halo.

```
41
42 void local2global_cumsum_x(double *scaf, int halo_method, map *M)
43
```

local2global_cumsum_x(...) revised on 23/05/2018 by Arnau Prat Gasull.

Description

Computes the global cumulative sum from relative cumulative scafs inthe x direction, analogously to local2global_cumsum_y(...). It is used at the core of cumsum_x(...).

Parameters

Below follows a brief description of the variables:
double *scaf (input/output) The scalar field which will be updated with the cumulative sum.
int halo_method (input) If 0, the cumulative sum is done excluding the halos. If 1, they are included. If 2 the cumulative sum is done on the domain points but in a fashion that gives the same results as if halo_update_y(...) had been called.
map *M (input) The scaf's initrinsic properties.

Part V

First program

APPENDIX A

Visualization of the results of First program

During the development of the First program a great deal of time was put into representing the data for validation purposes. Because in First program the information is stored in M-by-N-by-T matrices, where each of the slices of T contains information at a given timestep[1].

A tool to rapidly visualize multiple plots was created and its source code can be read in `plotSavedMAPS.m`, included below.

Multiple colormaps were created for this tool and these can be seen at `custom_colormaps.m`. The HSB Color Model had to be studied [1] in order to create realistic-enough colourmaps easily. The perceptually uniform colourmaps [11] of the `cmocean(...)` function body[2] were also added to the file.

[1] While storing the results in memory is not a good practice, it is a loss of time to develop a tool that exports the results to a file in Matlab, as syntax changes from Matlab to C.

[2] The function was downloaded from `https://mathworks.com/matlabcentral/fileexchange/57773-cmocean-perceptually-uniform-colormaps?s_tid=gn_loc_drop`.

APPENDIX B

Program listings

In this chapter, files of First program source code are presented. These functions have not been implemented the same way in Shallow Worlds but can be used as references for future improvements.

B.1 Domain's halo update tool

The halo update operation for sequential programs is much easier than the `halo_update(...)` operation presented in Shallow Worlds. Here, the halo is updated according to a periodic BC both in the x and y directions.

```
% FILE haloUpdate.m
% ==================
%
% DESCRIPTION
% -----------
% Ensures that a domain variable/field is periodic throughout the domain. MAP is the
    matrix containing the values of the variable.
%
% INPUTS
% ------
% - MAP: the matrix containing the values to be updated
%
% OUTPUTS
% -------
% - MAP: the matrix which satisfiest the periodic boundary condition
%

function MAP = haloUpdate(MAP)

    % Get matrix size
    [M, N] = size(MAP);

    % Ensure notation consistency between files
        M = M - 4;
    N = N - 4;

    % Ensure that the variable is periodic throughout the domain
    MAP(1, :) = MAP(M + 1, :);
    MAP(2, :) = MAP(M + 2, :);
    MAP(M + 3, :) = MAP(3, :);
    MAP(M + 4, :) = MAP(4, :);
    MAP(:, 1) = MAP(:, N + 1);
    MAP(:, 2) = MAP(:, N + 2);
    MAP(:, N + 3) = MAP(:, 3);
    MAP(:, N + 4) = MAP(:, 4);
```

```
end
```

B.2 Validate tool

The program that compiled all the functions that validated the results of First program is presented in this section.

```
clear;
close all;
clc;

adv_P1Check = 0;
adv_P2Check = 0;
adv_Check = 0;
cons_hCheck = 0;
AdamsBashforth_Check = 1;

T = 400; % Time steps
DeltaT = 1e-3; % [s]
t0 = 0; % Start (eval) time

NN = [10 20 40 80]; % Number of points per side

if adv_P1Check

    fprintf('CHECK THE FIRST PART OF THE ADVEDCTION\n');
    fprintf('Calling the function in adv_P1Error.m\n');

    for i=1:length(NN)
        [max_err_du(i), max_err_dv(i)] = adv_P1Error(NN(i), DeltaT);
    end

    figure;

    loglog(1./NN, max_err_du,'or-');
    hold on;
    loglog(1./NN, max_err_dv,'og-');
    loglog(1./NN, 1e-7./NN, 'ok-');
    loglog(1./NN, 3e-6./(NN.^2), 'ok-');

end

if adv_P2Check

    fprintf('CHECK THE SECOND PART OF THE ADVEDCTION\n');
    fprintf('Calling the function in adv_P2Error.m\n');

    for i=1:length(NN)
        [max_err_du(i), max_err_dv(i)] = adv_P2Error(NN(i));
    end

    figure;

    loglog(1./NN, max_err_du,'or-');
    hold on;
    loglog(1./NN, max_err_dv,'og-');
    loglog(1./NN, 8./NN.^2, 'ok-');
end

if adv_Check
end

if cons_hCheck

    fprintf('CHECK THE FIRST PART OF THE ADVEDCTION\n');
    fprintf('Calling the function in cons_hError.m\n');

```

```
    for i=1:length(NN)
        [max_err_dh(i)] = cons_hError(NN(i), DeltaT);
        % TODO es podria fer que la funcio sigui un argument
    end

    figure;

    loglog(1./NN, max_err_dh, 'or-');
    hold on;
    loglog(1./NN, 1e-7./NN, 'ok-');
    loglog(1./NN, 3e-6./(NN.^2), 'ok-');

end

if AdamsBashforth_Check

    [eta_n, eta_s] = coreError(NN(1), T, DeltaT);

    % Get colormaps
    custom_colormaps

    plotSavedMAPS({eta_n, eta_s}, {water, ground}, {'numerical', 'theoretical'}, 'time-
        integration', 20);

end
```

B.3 Validation using The Method of Manufactured Solutions

In this section, the functions used to validate Prototype and First program are presented.

```
% FILE cons_hError.m
% ==================
%
% BRIEF DESCRIPTION
% -----------------
% Verifies that the operations that compute the surface perturbations are correct.
%
% TECHNICAL DESCRIPTION
% ---------------------
% <TeX>For an arbitrary u = u(x,y,t), v = v(x,y,t) and h = h(x,y,t) defined in the body of the
    function, cons_hError.m computes the symbolic expression of f_h\footnote{Note that f_h
    is the source terms typically found in the equation, and in any case related to the
    Coriolis parameter f.} that fullfill \frac{\partial h}{\partial t} = -\frac{\partial (uh)}{x} - \frac{\partial (vh)}{y} + f_h. The function then takes
    the values of the expressions at discretized points and compares them with the
    values obtained with \texttt{cons_h.m}. \texttt{max_err_dh} is the difference
    between the theoretical values and the value obtained with the programmed function
    .</TeX>
%
% NOTES
% -----
% See adv_P1Error.m for the notes.
%
% INPUTS
% ------
% - NN: the number of points per side of the domain
% - DeltaT: the time increment between time iterations
%
% OUTPUTS
% -------
% - max_err_dh: the difference between the theoretical value and the  value obtained
    with the programmed function
%


function [max_err_dh] = cons_hError(NN, DeltaT)

    % Set the situation up
    M = NN; % Number of points in x direction
```

```
    N = NN; % Number of points in y direction
    L = 1; % [m] - Length of the sides
    t0 = 1; % [s] - Eval time

    % PART 1: ANALYTIC SOLUTION
    % -------------------------

    % Define symbolic variables
    syms x y t u_s v_s h_s;

    % Initialize symbolic fields with arbitrary periodic functions
    u_s = sin(2 * pi * x/L) * cos(2 * pi * y / L) * t;
    v_s = sin(2 * 2 * pi * x/L) * sin(2 * pi * y / L) * t;
    h_s = cos(2 * pi * x/L) * cos(2 * pi * y / L) * t;

    % Obtain the time derivative
    dh_dt = symfun( simplify( diff(h_s, t) ), [x y t] );

    % Convert symbolic expression to matlab function
    f_u_s = matlabFunction(u_s,'Vars',[x y t]); % u field
    f_v_s = matlabFunction(v_s,'Vars',[x y t]); % v field
    f_h_s = matlabFunction(h_s,'Vars',[x y t]); % h field
    f_dh_dt_s = matlabFunction(dh_dt,'Vars',[x y t]); % dh/dt field

    % Get staggered and centered coordinates
    mesh = createMesh(M, N, L);

    % Compute numeric values from the symbolic solution
    for j = 1:N + 4
        for  i = 1:M + 4

            % Compute the values of velocities according to the initialized fields
            u(i, j) = f_u_s(mesh.sx(i), mesh.cy(j), t0);
            v(i ,j) = f_v_s(mesh.cx(i), mesh.sy(j), t0);
            h(i ,j) = f_h_s(mesh.cx(i), mesh.cy(j), t0);

            % Compute h increment using the symbolic expression
            sym_dh(i, j) = f_dh_dt_s(mesh.cx(i), mesh.cy(j), t0) * DeltaT;

        end
    end

    % Ensure periodicity on the domain
    u = haloUpdate(u);
    v = haloUpdate(v);
    h = haloUpdate(h);

    % Display time step
    fprintf('DeltaT: %4.3e\n', DeltaT );

    % PART 2: NUMERIC SOLUTION
    % ------------------------

    % Compute source term
    cons_h_s = simplify( diff(h_s*u_s,x)+ diff(h_s*v_s,y) );

    % Compute source term at points
    fh_s = symfun( simplify( diff(h_s,t)+cons_h_s ) , [x,y,t] );

    % Convert symbolic expression to Matlab function
    f_fh_s = matlabFunction(fh_s,'Vars',[x y t]);

    % Compute numeric values for the symbolic source term
    for j = 1:N + 4
        for  i = 1:M + 4

            % Compute the source term
            fh(i, j) = f_fh_s(mesh.cx(i), mesh.cy(j), t0);

        end
```

```
    end

    % Ensure periodicity on the domain
    fh = haloUpdate(fh);

    % Compute the results of the advection using the numeric operations
    num_deltah = cons_h(u, v, h, L, DeltaT)  + fh*DeltaT;

    % Ensure periodicity on the domain
    num_deltah = haloUpdate(num_deltah);

    % PART 3 - COMPARISON
    % -------------------

    % Compare symbolic and numeric results
    err_dh = abs(num_deltah - sym_dh );

    % Get maximum error
    max_err_dh = max(abs(err_dh(:) )) ;

    % Print result
    fprintf('max_err_dh: %4.3e\n', max_err_dh );

end
```

```
% FILE adv_P1Error.m
% ==================
%
% BRIEF DESCRIPTION
% -----------------
% Verifies that the operations that compute the first part of the advection are correct.
%
% TECHNICAL DESCRIPTION
% ---------------------
% <TeX>For an arbitrary $u = u(x, y, t)$ and $v = v(x, y, t)$ defined in the body of the function, \texttt{adv\_P1Error.m} computes the symbolic expressions of $f_u$ and $f_v$\footnote{Note that $f_u$ and $f_v$ are the source terms typically found in the conservation of momentum equation, and in any case related to the Coriolis parameter $f$.} that fullfill $\frac{\partial u}{\partial t} = -\frac{\partial (uu)}{x} - \frac{\partial (uv)}{y} + f_u$ and $\frac{\partial u}{\partial t} = -\frac{\partial (vu)}{x} - \frac{\partial (vv)}{y} + f_v$. The function then takes the values of the expressions at discretized points and compares them with the values obtained with \texttt{adv\_Pu1.m} and \texttt{adv\_Pv1.m}. \texttt{max\_err\_du} and \texttt{max\_err\_dv} are the difference between the theoretical values and the values obtained with the programmed functions.</TeX>
%
% NOTES
% -----
% - symfun(..., [x y t]) ensures that the function is dependent on x, y and t.
% - matlabFunction(..., 'Vars', [x y z]) ensures that the function that is created takes the arguments in the [x y z] specified order.
%
% INPUTS
% ------
% - NN: the number of points per side of the domain
% - DeltaT: the time increment between time iterations
%
% OUTPUTS
% -------
% - max_err_du, max_err_dv: the difference between the theoretical values and the values obtained with the programmed functions
%

function [max_err_du, max_err_dv] = adv_P1Error(NN, DeltaT)

    % Set the situation up
    M = NN; % Number of points in x direction
    N = NN; % Number of points in y direction
    L = 1; % [m] - Length of the sides
    t0 = 1; % [s] - Eval time

    % PART 1: ANALYTIC SOLUTION
```

```
    % ------------------------

    % Define symbolic variables
    syms x y t u_s v_s;

    % Initialize symbolic fields with arbitrary periodic functions
    u_s = sin(2 * pi * x/L) * cos(2 * pi * y/L) * t;
    v_s = sin(2 * 2 * pi * x/L) * sin(2 * pi * y/L) * t;

    % Obtain the time derivative
    du_dt = symfun( simplify( diff(u_s, t) ), [x y t] );
    dv_dt = symfun( simplify( diff(v_s, t) ), [x y t] );

    % Convert symbolic expressions to Matlab functions
    f_u_s = matlabFunction(u_s,'Vars',[x y t]); % u field
    f_v_s = matlabFunction(v_s,'Vars',[x y t]);
    f_du_dt_s = matlabFunction(du_dt,'Vars',[x y t]); % du/dt field
    f_dv_dt_s = matlabFunction(dv_dt,'Vars',[x y t]);

    % Get staggered and centered coordinates
    mesh = createMesh(M, N, L);

    % Compute numeric values from the symbolic solution
    for j = 1:N + 4
        for  i = 1:M + 4

            % Compute the values of velocities according to the initialized fields
            u(i, j) = f_u_s(mesh.sx(i), mesh.cy(j), t0);
            v(i ,j) = f_v_s(mesh.cx(i), mesh.sy(j), t0);

            % Compute velocity increment using the symbolic expression
            sym_du(i, j) = f_du_dt_s(mesh.sx(i), mesh.cy(j), t0) * DeltaT;
            sym_dv(i, j) = f_dv_dt_s(mesh.cx(i), mesh.sy(j), t0) * DeltaT;

        end
    end

    % Ensure periodicity on the domain
    u = haloUpdate(u);
    v = haloUpdate(v);

    % Display time step
    fprintf('DeltaT: %4.3e\n', DeltaT );

    % PART 2: NUMERIC SOLUTION
    % ------------------------

    % Compute source terms
    p1u = simplify( diff(u_s*u_s,x)+ diff(u_s*v_s,y) );
    p1v = simplify( diff(u_s*v_s,x)+ diff(v_s*v_s,y) );

    % Compute source terms at points
    fu_s = symfun( simplify( diff(u_s,t)+p1u ), [x,y,t] );
    fv_s = symfun( simplify( diff(v_s,t)+p1v ), [x,y,t] );

    % Convert symbolic expressions to Matlab functions
    f_fu_s = matlabFunction(fu_s, 'Vars', [x y t]);
    f_fv_s = matlabFunction(fv_s, 'Vars', [x y t]);

    % Compute numeric values for the symbolic source term
    for j = 1:N + 4
        for  i = 1:M + 4

            % Compute the source terms
            fu(i, j) = f_fu_s(mesh.sx(i), mesh.cy(j), t0);
            fv(i ,j) = f_fv_s(mesh.cx(i), mesh.sy(j), t0);

        end
    end

```

```
    % Ensure periodicity on the domain
    fu = haloUpdate(fu);
    fv = haloUpdate(fv);

    % Compute the results of the advection using the numeric operations
    num_deltau = adv_Pu1(u, v, L, DeltaT)  + fu*DeltaT;
    num_deltav = adv_Pv1(u, v, L, DeltaT)  + fv*DeltaT;

    % Ensure periodicity on the domain
    num_deltau = haloUpdate(num_deltau);
    num_deltav = haloUpdate(num_deltav);

    % PART 3 - COMPARISON
    % -------------------

    % Compare symbolic and numeric results
    err_du = abs(num_deltau - sym_du);
    err_dv = abs(num_deltav - sym_dv);

    % Get maximum errors
    max_err_du = max(abs(err_du(:))) ;
    max_err_dv = max(abs(err_dv(:))) ;

    % Print results
    fprintf('max_err_du: %4.3e\n', max_err_du);
    fprintf('max_err_dv: %4.3e\n', max_err_dv);

end
```

```
% FILE adv_P2Error.m
% ==================
%
% BRIEF DESCRIPTION
% -----------------
% Verifies that the operations that compute the second part of the advection are correct
    .
%
% TECHNICAL DESCRIPTION
% ---------------------
% <TeX>Similarly to \texttt{adv\_P1Error.m}, this function compares the result of \
    texttt{adv\_Pu2.m} and \texttt{adv\_Pv2.m} with the theoretical value by taking
    advantage of the symbolic toolbox in Matlab. </TeX>
%
% NOTES
% -----
% - symfun(..., [x y t]) ensures that the function is dependent on x y t.
%
% INPUTS
% ------
% - NN: the number of points per side of the domain
%
% OUTPUTS
% -------
% - max_err_du, max_err_dv: the difference between the theoretical values and the
    values obtained with the programmed functions
%
function [max_err_du, max_err_dv] = adv_P2Error(NN)

    % Set the situation up
    M = NN; % Number of points in x direction
    N = NN; % Number of points in y direction
    L = 1; % [m] - Length of the sides

    % PART 1: ANALYTIC SOLUTION
    % -------------------------

    % Define symbolic variables
    syms x y;

    % Initialize symbolic fields with arbitrary periodic functions
    u_s = sin( (x / L) * 2 * pi ) + cos( (y / L) * 2 * pi );
```

```
    v_s = cos( (x / L) * 2 * pi ) + sin( (y / L) * 2 * pi );

    % Obtain the symbolic expressions for advection
    du_s = simplify(u_s * ( diff(u_s, x) + diff(v_s, y) ));
    dv_s = simplify(v_s * ( diff(u_s, x) + diff(v_s, y) ));

    % Obtain the symbolic expressions for advection
    du_s = simplify(u_s * ( diff(u_s, x) + diff(v_s, y) ));
    dv_s = simplify(v_s * ( diff(u_s, x) + diff(v_s, y) ));

    % Convert symbolic expression to Matlab functions
    f_u_s = matlabFunction(u_s,'Vars',[x y]);
    f_v_s = matlabFunction(v_s,'Vars',[x y]);
    f_du_s = matlabFunction(du_s,'Vars',[x y]);
    f_dv_s = matlabFunction(dv_s,'Vars',[x y]);

    % Get staggered and centered coordinates
    mesh = createMesh(M, N, L);

    % Compute symbolic
    for j = 1:N + 4
        for  i = 1:M + 4

            % Compute the values of velocities according to the initialized fields
            u(i, j) = f_u_s(mesh.sx(i), mesh.cy(j));
            v(i ,j) = f_v_s(mesh.cx(i), mesh.sy(j));

            % Compute the results of the advection using the symbolic expression
            sym_du_P2(i, j) = f_du_s(mesh.sx(i), mesh.cy(j));
            sym_dv_P2(i, j) = f_dv_s(mesh.cx(i), mesh.sy(j));

        end
    end

    % Ensure periodicity on the domain
    u = haloUpdate(u);
    v = haloUpdate(v);

    % PART 2: NUMERIC SOLUTION
    % ------------------------

    % Compute the results of the advection using the numeric operations
    num_du_P2 = adv_Pu2(u, v, L);
    num_dv_P2 = adv_Pu2(u, v, L);

    % Update halo
    num_du_P2 = haloUpdate(num_du_P2);
    num_dv_P2 = haloUpdate(num_du_P2);

    % Compare symbolic and numeric results
    err_du = abs(num_du_P2 - sym_du_P2);
    err_dv = abs(num_du_P2 - sym_du_P2);

    % Get maximum errors
    max_err_du = max(err_du(:));
    max_err_dv = max(err_dv(:));

    % Print results
    fprintf('max_err_du: %4.3e\n', max_err_du);
    fprintf('max_err_dv: %4.3e\n', max_err_dv);

end
```

B.4 Plotting tools

The plotting tools are an important part of First program. A great effort was put into understanding the theory behind colormaps and the inner workings of Matlab's plotting suite.

```
% FILE plotSavedMAPS.m
% ====================
%
% DESCRIPTION
% -----------
% Returns an animated plot of the variables specified in MAPS, i.e. MAPS = {u_save
    eta_save} with the colormap specified in COLORMAPS, i.e. COLORMAPS = {blues reds}.
    The legend is specified in TAGS, i.e. TAGS  = {'key 1', 'key 2'} while the filename
    is specified in filename
%
% INPUTS
% ------
% - MAPS: the cell array containing the snapshots of the variable at a given timestep
% - COLORMAPS: the cell array containing the matrices with the colors for each z level (
    see surf2colormap.m)
% - TAGS: the cell array containing the strings of the legend
% - filename: the name of the file to be produced
%

function plotSavedMAPS(MAPS, COLORMAPS, TAGS, filename, plot_period)

    % Import colormaps
    custom_colormaps;

    % Import variables that are often used
    inputs;

    % Determine the number of MAPs to be plotted and passed through MAPS
    num_MAPS = length(MAPS);

    figure;
    for t = 1:plot_period:T

        for m = 1:num_MAPS

            % Get one the elements of the cell arrays
            MAP = MAPS{m};
            COLORMAP = COLORMAPS{m};

            % Get matrix size
            [M, N, T] = size(MAP);

            % Asssign colormap
            RGB_COLORMAP = surf2colormap(MAP, COLORMAP, T);

            % Ensure notation consistency between files
            M = M - 4;
            N = N - 4;

            % Compute maximum and minimum of the MAP
            lower_limit = min(min(min(MAP)));
            upper_limit = max(max(max(MAP)));

            % Set limits if ther is an error
            if lower_limit == upper_limit
                upper_limit = 2;
                lower_limit = 0;
            end

            % Print limits
            % fprintf('lower_limit = %d\nupper_limit = %d\n', [lower_limit upper_limit])
                ;

            % permute(RGB_COLORMAP(:, :, t, :), [2 1 4 3]) returns the COLORMAP aligned
                with transpose(MAP(:,:,t))
            surf(transpose(MAP(:,:,t)), permute(RGB_COLORMAP(:, :, t, :), [2 1 4 3]));

            shading interp

            % Set viewpoint
```

```
            view(-37.5, 70);

            % Set axis limits
            xlim([1, M+4]);
            ylim([1, N+4]);
            zlim([lower_limit upper_limit]);

            % Set title
            title(filename);

            hold on
        end

        % Add legend
        legend(TAGS);

        hold off

        % Plot at every iteration
        drawnow;

        % Store the plot
        frame = getframe(1);
        curr_frame = 1;
        im{curr_frame} = frame2im(frame);

        % Print status
        fprintf('Frame #%d plotted\n', t);

        % Write GIF file

        [A, map] = rgb2ind(im{curr_frame}, 256);
        if t == 1
            imwrite(A, map, filename, 'gif', 'LoopCount', Inf, 'DelayTime', 0);
        else
            imwrite(A, map, filename, 'gif', 'WriteMode', 'append', 'DelayTime', 0);
        end

        curr_frame = curr_frame + 1;

        % Print status
        fprintf('Frame #%d saved\n', t);

    end
    hold off

    % Create GIF
    % [a, b]  = findIntegerFactorsCloseToSquareRoot(t);

    % figure;
    % for t = 1:T
    %
    % %    subplot(b, a, t);
    % %    imshow(im{t});
    %     [A, map] = rgb2ind(im{t}, 256);
    %
    %     if t == 1
    %         imwrite(A, map, filename, 'gif', 'LoopCount', Inf, 'DelayTime', 0);
    %     else
    %         imwrite(A, map, filename, 'gif', 'WriteMode', 'append', 'DelayTime', 0);
    %     end
    %
    %     fprintf('Frame #%d saved\n', t);
    % end
end
```